KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

YERÇEKİMSEL N-CİSİM SİMÜLASYONUNDAKİ KARŞILIKLI KUVVETLER OPTİMİZASYONUNUN GPU ÜZERİNDEKİ ANALİZİ

YÜKSEK LİSANS TEZİ

Bilgisayar Müh. Celil ÖZKURT

HAZİRAN 2018
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce

Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih : / /

Tezin Savunma Tarihi : / /

Tez Danışmanı :

Trabzon

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

başlıklı bu çalışma, Enstitü Yönetim Kurulunun / / gün ve sayılı kararıyla oluşturulan jüri tarafından yapılan sınavda
YÜKSEK LİSANS TEZİ
olarak kabul edilmiştir.

Jüri Üyeleri

Başkan : …...……………………....………

Üye : …...…………....…………………

Üye : ……...……………...……………

Prof. Dr. Sadettin KORKMAZ

Enstitü Müdürü

ÖNSÖZ

Paralel programlama, günümüzde veri boyutlarının büyümesiyle her geçen gün daha popüler olmaktadır. GPU'ların hesaplama gücünün artmasıyla birlikte paralel programlama grafik kartları ile gerçekleştirebilir hale gelmiştir. Ayrıca, grafik kartları seri üretim donanımlar olduğundan paralel programlamayı özel şartlarda ve ortamlarda gerçekleştirilebilen bir yöntem olmaktan herkesin ulaşabileceği bir yönteme dönüştürmüştür.

Tez çalışmasında, Nvidia CUDA paralel hesaplama mimarisi ve bu mimarinin sunduğu programlama dili kullanılarak, Yerçekimsel N-cisim simülasyonundaki Karşılıklı kuvvetler optimizasyonunun GPU üzerinde yürütme zamanı, hesaplama gücü ve bellek kullanımı gibi temel kriterlere göre analizinin yapılması amaçlanmıştır. Analizler Yerçekimsel N-cisim simülasyonundaki Bütün-çiftler yöntemi ile karşılaştırılarak verilmiştir.

Tez konumun belirlenmesinde ve çalışmalarımın her aşamasında yardımlarını esirgemeyip değerli fikir ve katkılarıyla ışık tutan ve yönlendiren, tez metinini inceleyerek biçim ve içerik bakımından son şeklini almasına katkıda bulunan danışmanım Sayın Dr. Öğr. Üyesi Eyüp GEDİKLİ hocama sonsuz teşekkürlerimi sunarım. Ayrıca her anımda yanımda olan sevgili eşim Selma ÖZKURT'a teşekkür ederim.

Celil ÖZKURT
Trabzon 2018

TEZ ETİK BEYANNAMESİ

Yüksek Lisans Tezi olarak sunduğum "Yerçekimsel N-Cisim Simülasyonundaki Karşılıklı Kuvvetler Optimizasyonunun GPU Üzerindeki Analizi" başlıklı bu çalışmayı baştan sona kadar danışmanım Dr. Öğr. Üyesi Eyüp GEDİKLİ'nin sorumluluğunda tamamladığımı, verileri/örnekleri kendim topladığımı, deneyleri/analizleri ilgili laboratuvarlarda yaptığımı/yaptırdığımı, başka kaynaklardan aldığım bilgileri metinde ve kaynakçada eksiksiz olarak gösterdiğimi, çalışma sürecinde bilimsel araştırma ve etik kurallara uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim. 07/06/2018

Celil ÖZKURT

İÇİNDEKİLER

Sayfa No

ÖNSÖZ III

TEZ ETİK BEYANNAMESİ IV

İÇİNDEKİLER V

ÖZET VIII

SUMMARY IX

ŞEKİLLER DİZİNİ X

TABLOLAR DİZİNİ XII

SEMBOLLER DİZİNİ XIII

1 GENEL BİLGİLER 1

1.1. Giriş 1

1.2. Paralel Hesaplama 1

1.2.1. Seri ve Paralel Programlama 3

1.2.2. Paralellik 4

1.2.3. Bilgisayar Mimarisi 5

1.3. Heterojen Hesaplama 7

1.3.1. Heterojen Mimari 7

1.3.2. Heterojen Hesaplama Paradigması 9

1.3.3. CUDA: Heterojen Hesaplama İçin Bir Mimari 10

1.4. CUDA 13

1.4.1. CUDA Programlama Modeli 13

1.4.2. CUDA Bellek Modeli 18

1.4.3. CUDA Akışlar (Streams) 23

1.4.4. CUDA Uygulamasında Dikkat Edilmesi Gereken Noktalar 27
1.5. N-Cisim Simülasyonu 34
1.5.1. N-Cisim Simülasyonundaki Karşılıklı Kuvvetler 35
1.5.2. Literatür Araştırması.. 36
2. YAPILAN ÇALIŞMALAR.. 38
2.1. Giriş 38
2.2. Bütün-Çiftler(All-Pairs) Yönteminin CUDA ile Gerçekleştirilmesi......................... 38
2.3. Karşılıklı Kuvvetler Optimizasyonunun CUDA ile Gerçekleştirilmesi 39
2.4. KKO'nun CUDA Stream Özelliği Kullanılarak Gerçekleştirilmesi 40
2.5. KKO'da Bloklar Arası İş Yükünün Düzenlenmesi 42
2.6. Algoritmaların Temel Optimizasyon Yöntemleri ile Optimize Edilmesi 44
2.6.1. Farklı Yapıdaki Grid ve Blok Yapısı.. 44
2.6.2. Paylaşımlı Bellek Kullanımı.. 45
2.6.3. Döngü Açma (Loop Unrolling) Tekniği.. 46
3. BULGULAR VE TARTIŞMA.. 47
3.1. Giriş 47
3.2. Çalışma Zamanına Göre Algoritmaların Karşılaştırılması....................................... 47
3.3. Streams Özelliğinin Kullanımına Göre Algoritmalarının Karşılaştırılması.............. 48
3.4. FLOP/s Değerine Göre Algoritmaların Karşılaştırılması... 48
3.5. Bellek Kullanımlarına Göre Algoritmaların Karşılaştırılması 50
3.6. Erişilen Thread Kullanım Oranına Göre Algoritmaların Karşılaştırılması 52
3.7. SM Kullanım Oranına Göre Algoritmaların Karşılaştırılması 53
3.8. Farklı Grid ve Blok Yapılarına Göre Algoritmalarının Karşılaştırılması.................. 55
3.9. Paylaşımlı Bellek Kullanımına Göre BÇA'nın Karşılaştırılması............................. 55
3.10. Döngü Açma Optimizasyonuna Göre Algoritmalarının Karşılaştırılması 57
4. SONUÇLAR.. 60

5. ÖNERİLER 61
6. KAYNAKLAR 62
ÖZGEÇMİŞ

Yüksek Lisans Tezi

ÖZET

YERÇEKİMSEL N-CİSİM SİMÜLASYONUNDAKİ KARŞILIKLI KUVVETLER OPTİMİZASYONUNUN GPU ÜZERİNDEKİ ANALİZİ

Celil ÖZKURT

Karadeniz Teknik Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Dr. Öğr. Üyesi Eyüp GEDİKLİ
2018, 66 Sayfa

Bu tez kapsamında, yerçekimsel N-cisim simülasyonundaki karşılıklı kuvvetler optimizasyonu CUDA paralel hesaplama mimarisi kullanılarak GPU üzerinde uygulanıp, kullanılan algoritmanın analizi yapılmıştır. N-cisim problemlerinin çözümü, çok sayıda hesaplama gerektirmekte ve uzun zaman almaktadır. Çalışmada, GPU ile bu tarz problemlerin çözümü için daha hızlı yöntemlerin üretilmesi amaçlanmıştır. Algoritmanın analizinde kullanılan önemli metrikler yürütme zamanı, hesaplama gücü ve bellek kullanımıdır. Analizler, GPU üzerinde CUDA ile uygulanan bütün-çiftler algoritması ile karşılaştırmalı olarak verilmiştir. Analizler sonucunda, veri büyüklüğü arttıkça, karşılıklı kuvvetler optimizasyonu bütün-çiftler algoritmasına karşı yürütme zamanı açısından daha iyi sonuç elde etmiştir. Hesaplama gücüne göre ise karşılıklı kuvvetler algoritması bütün-çiftler algoritmasına karşı yaklaşık 2 kat daha kötü sonuç elde etmiştir. Bellek kullanımı bakımından ise iki algoritma arasında denk bir kullanım vardır. Bu sonuçlar, veri setinin büyük olduğu simülasyonlarda, karşılıklı kuvvetler optimizasyonunun bütün-çiftler algoritmasının yerine kullanılabilir olduğunu göstermektedir.

Anahtar Kelimeler: Yerçekimsel N-cisim simülasyonu, Karşılıklı kuvvetler, CUDA, GPU, Yüksek-performanslı hesaplama, GPU ile paralel hesaplama

Master Thesis

SUMMARY

ANALYSIS OF THE OPTIMIZATION OF RECIPROCAL FORCES IN THE GRAVITATIONAL N-BODY SIMULATION ON GPU

Celil ÖZKURT

Karadeniz Technical University
The Graduate School of Natural and Applied Sciences
Computer Engineering Graduate Program
Supervisor: Dr. Lecturer Eyüp GEDİKLİ
2018, 66 Pages

In this thesis, the optimization of reciprocal forces in the gravitational N-body simulation is implemented on the GPU using CUDA parallel computing architecture and the algorithm used is analyzed. The solution of N-body problems requires a lot of computation and takes a long time. In the study, it was aimed to produce faster methods for solving such problems with GPU. Important metrics used in the analysis of the algorithm are execution time, computing power and memory usage. Analyzes are given in comparison with the all-pairs algorithm applied with CUDA on GPU. As a result of the analysis, as the data size increases, reciprocal forces optimization has achieved better results in terms of execution time against the all-pair algorithm. According to the calculation power, the reciprocal force algorithm is about 2 times worse than the all-pair algorithm. In terms of memory usage, there is an equivalent use between the two algorithms. These results show that in the simulations where the dataset is large, the reciprocal force optimization can be used instead of the all-pair algorithm.

Keywords: Gravitational N-body simulation, Reciprocal forces, CUDA, GPU, High-performance computing, Parallel computation with GPU

ŞEKİLLER DİZİNİ

Sayfa No

Şekil 1.1. Harvard mimarisi 2
Şekil 1.2. Seri programlama yapısı 3
Şekil 1.3. Paralel programlama yapısı 4
Şekil 1.4. Flynn taksonomisi 6
Şekil 1.5. CPU ve GPU arasındaki mimari farkı 8
Şekil 1.6. CPU ve GPU çalışma alanları 9
Şekil 1.7. CPU ve GPU'nun çalıştırdığı kod kısımları 10
Şekil 1.8. CUDA API ve programlama dilleri 11
Şekil 1.9. CUDA API seviyesi 12
Şekil 1.10. CUDA paralel hesaplama platformu 13
Şekil 1.11. Program ve program modelinin uygulaması arasındaki soyutlama 14
Şekil 1.12. Tipik bir CUDA uygulaması 16
Şekil 1.13. CUDA iş akışı 17
Şekil 1.14. CUDA thread hiyerarşisi 18
Şekil 1.15. Bellek hiyerarşisi 20
Şekil 1.16. CUDA bellek modeli 21
Şekil 1.17. Seri (eşzamanlı olmayan) akışlar 24
Şekil 1.18. Eşzamanlı akışlar 25
Şekil 1.19. Kaynaklar yetmediğinde akışların davranışı 26
Şekil 1.20. Thread açısından çoklu stream yapısı 26
Şekil 1.21. CUDA uygulama geliştirme süreci 27
Şekil 1.22. CUDA programının bağımlılık analizi 30
Şekil 1.23. CUDA çekirdek (kernel) performansını kısıtlayıcı faktörler 31
Şekil 1.24. GPU'da akışlı çoklu işlemci (streaming multiprocessor - SM) kullanımı 31
Şekil 1.25. Thread, warp, kaydedici ve paylaşımlı bellek kullanımı 32
Şekil 1.26. Çekirdeğin çalışması boyunca GPU birimlerinden faydalanma oranı 32
Şekil 1.27. Bellek türlerinin kullanımı 33
Şekil 1.28. Çalıştırılan komutların toplam komut sayısına oranları 33
Şekil 1.29. İki cisim arasındaki çekim kuvveti 35

Şekil 2.1. Bütün-çiftler yönteminin CUDA ile gerçekleştirilmesi 38
Şekil 2.2. Karşılıklı kuvvetler optimizasyonunun CUDA ile uygulaması 40
Şekil 2.3. Streams kullanılmadığında grid ve blok yapısı 41
Şekil 2.4. Streams kullanıldığında grid ve blok yapısı 41
Şekil 2.5. Karşılıklı kuvvetler optimizasyonunda blokların çalışma zamanları 42
Şekil 2.6. Karşılıklı kuvvetler optimizasyonunda blokların düzenlenmesi 43
Şekil 2.7. Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonun çalışma zamanı 43
Şekil 2.8. 24 x 512 thread organizasyonu 44
Şekil 2.9. GPU'da paylaşımlı bellek gösterimi 45
Şekil 2.10. Faktörü 2 olan döngü açma 46
Şekil 3.1. Algoritmaların çalışma zamanı 48
Şekil 3.2. Streams kullanımına göre KKO 49
Şekil 3.3. Streams kullanımına göre BDKKO 49
Şekil 3.4. Algoritmaların FLOP/s karşılaştırması 50
Şekil 3.5. Algoritmaların FLOP karşılaştırması 50
Şekil 3.6. Algoritmaların L1 önbellek kullanımı 51
Şekil 3.7. Algoritmaların L2 önbellek kullanımı 51
Şekil 3.8. Algoritmaların global bellek kullanımı 52
Şekil 3.9. Algoritmaların erişilen thread kullanım oranı 53
Şekil 3.10. Algoritmaların çoklu işlemci kullanımı 54
Şekil 3.11. KKO'nun erişilen thread ve çoklu işlemci kullanımı 55
Şekil 3.12. BÇA'nın farklı grid ve blok yapısında karşılaştırılması 56
Şekil 3.13. KKO'nun farklı grid ve blok yapısında karşılaştırılması 56
Şekil 3.14. BDKKO'nun farklı grid ve blok yapısında karşılaştırılması 57
Şekil 3.15. BÇA'nın paylaşımlı bellek kullanımı 57
Şekil 3.16. BÇA'nın paylaşımlı bellek kullanımın L1 önbelleğe etkisi 58
Şekil 3.17. BÇA'nda döngü açma yöntemi 58
Şekil 3.18. KKO'nunda döngü açma yöntemi 59
Şekil 3.19. BDKKO'nunda döngü açma tekniği 59

TABLOLAR DİZİNİ

Sayfa No

Tablo 1.1 Hiyerarşik yöntemler ve karmaşıklıkları 36

SEMBOLLER DİZİNİ

ALU	Arithmetic Logic Unit
API	Application Programming Interface
BÇA	Bütün-Çiftler Algoritması
BDKKO	Blokları Düzenlenmiş Karşılıklı Kuvvetler Optimizasyonu
CPU	Central Processing Unit
ÇKÇV	Çoklu Komut Çoklu Veri
ÇKTV	Çoklu Komut Tek Veri
DRAM	Dynamic Random Access Memory
FLOPS	Floating Point Operations Per Second
GPU	Graphics Processing Unit
HPC	High Performance Computing
KKO	Karşılıklı Kuvvetler Optimizasyonu
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
NVCC	NVIDIA's CUDA Compiler
PCI	Peripheral Component Interconnect
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
SRAM	Static Random Access Memory
TKÇV	Tek Komut Çoklu Veri
TKTV	Tek Komut Tek Veri

1. GENEL BİLGİLER

1.1. Giriş

Yüksek performanslı hesaplama (High-performance computing - HPC) [1] ortamı yeni teknolojiler ve süreçler yaygınlaştıkça değişmektedir. Buna dayalı olarak yüksek performanslı hesaplamanın tanımı da değişmektedir. Yüksek performanslı hesaplama genel olarak, yüksek verimlilik ve verimlilik ile eşzamanlı olarak karmaşık bir görevi gerçekleştirmek için birden fazla işlemci veya bilgisayar kullanımı ile ilgilidir [2]. Yüksek performanslı hesaplamayı sadece bir bilgisayar mimarisi olarak değil, aynı zamanda donanım sistemleri, yazılım araçları, programlama platformları ve paralel programlama paradigmaları dâhil bir dizi unsur olarak düşünmek gerekmektedir.

Son on yılda, yüksek performanslı hesaplama, özellikle paralel programlamada temel bir paradigma kaymasına yol açan GPU-CPU heterojen mimarilerinin ortaya çıkması nedeniyle önemli ölçüde gelişmiştir.

1.2. Paralel Hesaplama

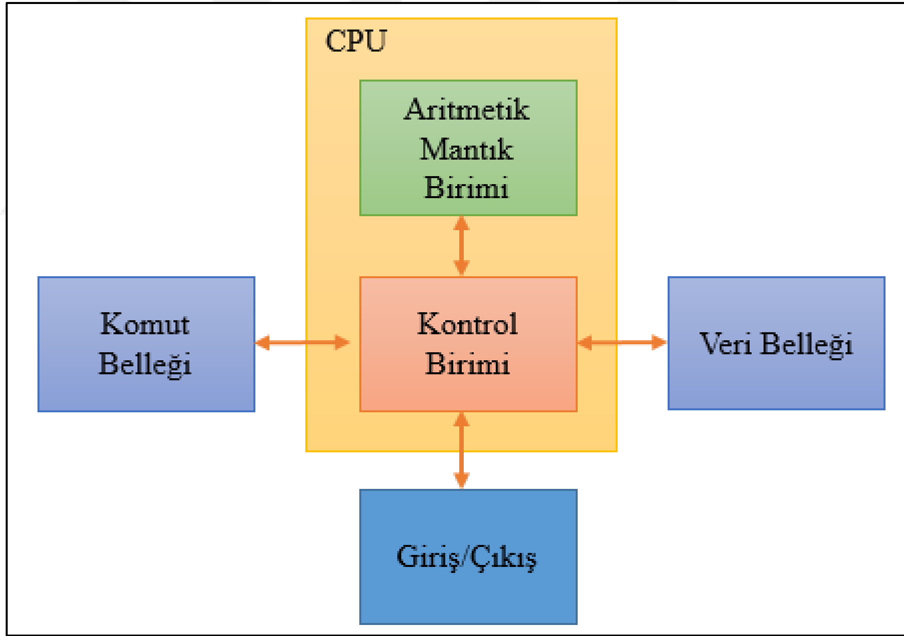
Paralel hesaplama son yıllarda gittikçe artan bir ilgi görmektedir. Paralel hesaplamanın amacı hesaplama hızını artırmaktır. Salt hesaplama bakış açısına göre paralel hesaplama (parallel computing): birçok hesaplamaların aynı anda gerçekleştirildiği bir hesaplama biçimi olarak tanımlanmaktadır [3]. Paralel hesaplama, genellikle büyük sorunların daha sonra çözülecek olan daha küçük olanlara bölünebileceği prensibine dayanarak çalışmaktadır. Programcının bakış açısına göre paralel hesaplama: eş zamanlı hesaplamaları bilgisayarlara nasıl eşleştirileceği anlamına gelmektedir. Birden fazla bilgi işlem kaynağının olduğu varsayıldığında, paralel hesaplama, eşzamanlı hesaplamaları gerçekleştirmek için çoklu hesaplama kaynaklarının (çekirdekler veya bilgisayarlar) eşzamanlı kullanımı olarak tanımlanmaktadır. Büyük bir problem daha küçük bölümlere bölünmekte ve daha küçük olan her problem farklı bilgi işlem kaynaklarında eşzamanlı olarak çözülmektedir. Paralel hesaplamanın yazılım ve donanım özellikleri birbiriyle iç içe bulunmaktadır. Aslında, paralel hesaplama iki farklı bilgisayar teknolojisini içermektedir:

- Bilgisayar mimarisi (donanım)
- Paralel programlama (yazılım)

Bilgisayar mimarisi, mimari seviyede paralelliği desteklemeye odaklanırken, paralel programlama, bilgisayar mimarisinin hesaplama gücünü tamamen kullanarak problemin çözümüne odaklanmaktadır. Yazılımda paralel yürütmeyi gerçekleştirebilmek için, donanımın, çoklu işlemin veya çoklu iş parçacığının (thread) eşzamanlı yürütülmesini destekleyen bir platform sağlaması gerekmektedir.

Çoğu modern işlemci, Şekil 1.1'de gösterildiği gibi üç ana bileşenden oluşan Harvard mimarisini uygulamaktadır [4]:

- Bellek (komut ve veri belleği)
- Merkezi işlem birimi (kontrol birimi ve aritmetik mantık birimi)
- Giriş/Çıkış ara yüzü



Şekil 1.1. Harvard mimarisi

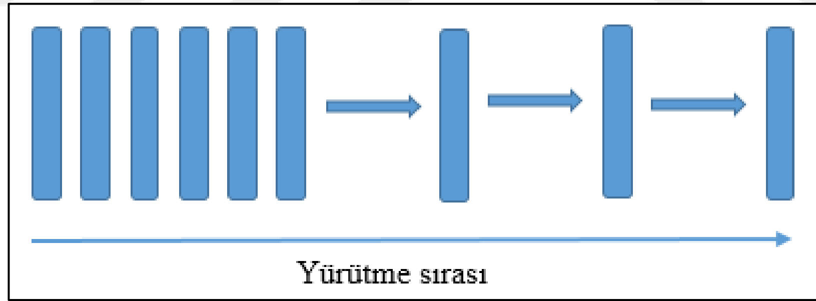
Yüksek performanslı bilgisayarlarda anahtar bileşen, genellikle çekirdek (core) olarak adlandırılan merkezi işlem birimidir (control processing unit - CPU) . Bilgisayarın ilk günlerinde bir yonga üzerinde yalnızca bir çekirdek bulunmaktaydı. Bu mimari tek işlemci (uniprocessor) olarak adlandırılmaktadır. Günümüzde yonga tasarımı eğilimi, mimari seviyede paralelliği desteklemek için birden fazla çekirdeği tek bir işlemci üzerine entegre

etme yönündedir. Bu mimari çok çekirdekli (multicore) olarak adlandırılmaktadır. Bu nedenle, programlama, problemin hesaplanmasını mevcut çekirdeklere eşleme süreci olarak görülmekte ve böylelikle paralel yürütme elde edilmektedir.

Sıralı bir algoritma uygularken, doğru bir program yazmak için bilgisayar mimarisinin ayrıntılarının anlaşılması gerekmemektedir. Bununla birlikte, çok çekirdekli makineler için algoritmalar uygularken, programcıların temel bilgisayar mimarisinin özelliklerinden haberdar olması çok daha önemlidir. Hem doğru hem de verimli paralel programlar yazmak çok çekirdekli mimariler hakkında temel bir bilgiye ihtiyaç duyulmaktadır.

1.2.1. Seri ve Paralel Programlama

Bir bilgisayar programıyla ilgili bir sorun çözülürken, sorunun ayrı bir dizi hesaplamaya bölünmesi doğaldır. Şekil 1.2.'deki gibi, her hesaplama belirli bir görevi yerine getirmektedir. Bu tarz programlar seri program (sequential program) olarak adlandırılmaktadır. Seri programlamada hesaplamalar ardışık olarak yapılır. Birinin başlaması için diğerinin bitmesi gerekir.



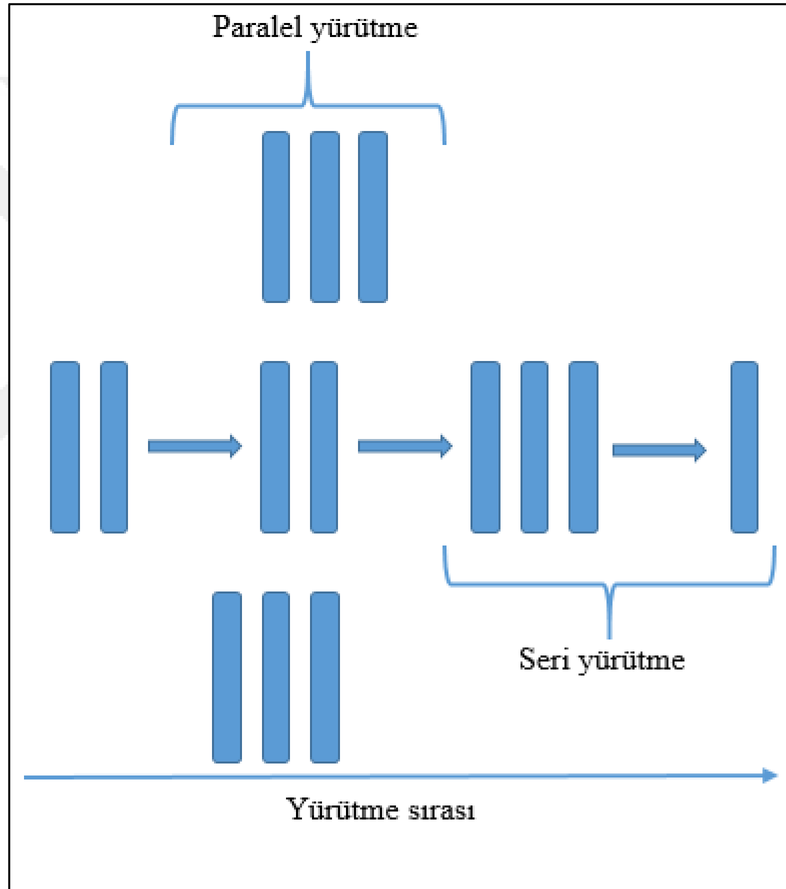
Şekil 1.2. Seri programlama yapısı

İki hesaplama arasındaki ilişki iki şekilde sınıflandırılmaktadır: Birincisinde öncelik kısıtlaması vardır ve bu nedenle seri bir şekilde hesaplanmalıdırlar. İkincisinde herhangi bir öncelik kısıtlaması yoktur ve eş zamanlı olarak hesaplanabilirler. Eşzamanlı olarak gerçekleştirilen görevleri içeren programlar paralel programdır. Şekil 1.3.'de gösterildiği gibi, paralel bir programda bazı parçalar seri olabilmektedir.

Bir programcının açısından, bir program iki temel bileşenden oluşmaktadır: komutlar (instruction) ve veri (data). Hesaplama problemi birçok küçük hesaplama parçasına

bölünmekte ve her parça bir görev (task) olarak adlandırılmaktadır. Bir görevde, bireysel komutlar girdileri işlemekte, bir fonksiyon uygulamakta ve çıktılar üretmektedir. Bir komut, bir önceki komut tarafından üretilen verileri işlediğinde bir veri bağımlılığı (data dependency) oluşmaktadır [5]. Bu nedenle, herhangi iki görev arasındaki ilişki bağımlı olarak, biri diğerinin çıktısını tüketirse veya bağımsız olarak sınıflandırılmaktadır.

Veri bağımlılıklarının analiz edilmesi, paralel algoritmaların uygulanmasında temel bir beceridir. Çünkü bağımlılıklar, paralellik için birincil önleyici (inhibitör) maddelerden biridir ve bunları anlamak uygulama hızını artırmak için gereklidir.



Şekil 1.3. Paralel programlama yapısı

1.2.2. Paralellik

Günümüzde, paralellik her yerde bulunmakta ve programlama dünyasında paralel programlama ana akım haline gelmektedir. Değişik seviyelerdeki paralellik, mimari tasarımın itici gücü olmaktadır. Uygulamalarda iki temel paralellik türü bulunmaktadır:

- Görev paralellik
- Veri paralellik

Görev paralelliği (task parallelism), birbirinden bağımsız ve büyük oranda paralel olarak çalıştırılabilecek pek çok görev veya fonksiyon olduğunda ortaya çıkmaktadır [6]. Görev paralelliği, fonksiyonları birden çok çekirdeğe dağıtmaya odaklanmaktadır.

Veri paralelliği (data parallelism), aynı anda çalıştırılabilecek çok sayıda veri öğesi bulunduğu zaman ortaya çıkmaktadır [7]. Veri paralelliği, verilerin birden çok çekirdeğe yayılmasına odaklanmaktadır. CUDA, NVIDIA'nın GPU (grafik işlem birimi) gücünü kullanarak hesaplama performansında büyük ölçüde artışlara olanak veren paralel hesaplama mimarisidir. Bugüne kadar piyasalarda ilgili alanlardaki CUDA; etkinleştirilmiş GPU ile görüntü ve video işleme, hesaplamaya dayalı biyoloji ve kimya, akışkan dinamiği, bilgisayarlı tomografi, sismik analiz, ışın izleme ve çok daha fazlası dahil olmak üzere geniş bir aralıkta kullanım alanları bulmaktadır.

1.2.3. Bilgisayar Mimarisi

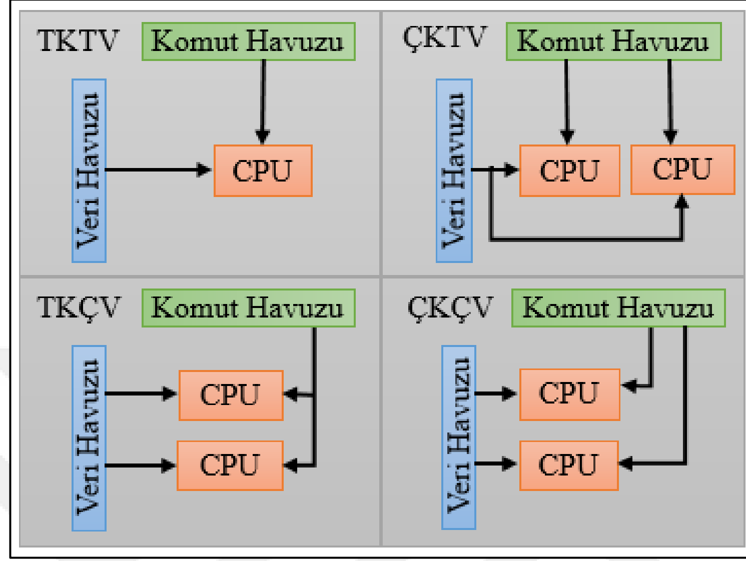
Bilgisayar mimarilerini sınıflandırmanın birkaç farklı yolu vardır. Flynn'in Taksonomisi [8], yaygın olarak kullanılan bir sınıflandırma şeması olup, mimarileri komutların ve verilerin çekirdeğin içinden nasıl aktığına göre Şekil 1.4.'deki gibi dört farklı biçimde sınıflandırmaktadır:

- Tek Komut Tek Veri (TKTV)
- Tek Komut Çoklu Veri (TKÇV)
- Çoklu Komut Tek Veri (ÇKTV)
- Çoklu Komut Çoklu Veri (ÇKÇV)

Tek Komut Tek Veri (Single Instruction Single Data - SISD) geleneksel bilgisayar anlamına gelmektedir. Seri bir mimarisi vardır ve bilgisayarda yalnızca bir çekirdek bulunmaktadır. Herhangi bir zamanda yalnızca bir komut akışı yürütülmekte ve işlemler sırasında yalnızca bir veri akışı gerçekleştirilmektedir.

Tek Komut Çoklu Veri (Single Instruction Multiple Data - SIMD) bir tür paralel mimari anlamına gelmektedir. Bilgisayarda birden çok çekirdek vardır. Tüm çekirdekler herhangi bir anda aynı komut akışını çalıştırmaktadır. Her biri farklı veri akışlarında çalışmaktadır. Vektörel bilgisayarlar tipik olarak Tek Komut Çoklu Veri olarak karakterize edilmekte ve en modern bilgisayarlar Tek Komut Çoklu Veri mimarisi kullanmaktadır. Belki

de Tek Komut Çoklu Verinin en büyük avantajı, CPU üzerinde kod yazarken programcıların seri bir program yazıyormuş gibi düşünmeye devam edebilmeleridir. Çünkü paralel hızlanma elde etmek için paralelleştirme işlemleri derleyici tarafından yapılmaktadır.



Şekil 1.4. Flynn taksonomisi

Çoklu Komut Tek Veri (Multiple Instruction Single Data - MISD), geleneksel mimariden farklı bir mimari ifade etmektedir. Bu mimaride, her bir çekirdek, ayrı komut akışları yoluyla aynı veri akışında çalışmaktadır.

Çoklu Komut Çoklu Veri (Multiple Instruction Multiple Data - MIMD) de bir tür paralel mimari anlamına gelmektedir. Çoklu çekirdekler birden fazla veri akışı üzerinde çalışmakta ve her biri bağımsız komutlar uygulamaktadır.

Çok çekirdekli (many core) terimi, genellikle yüksek sayıda (onlarca veya yüzlerce) çekirdeği olan çoklu çekirdekli (multicore) mimarileri tanımlamak için kullanılmaktadır. Son zamanlarda bilgisayar mimarilerinde, çoklu çekirdekten çok çekirdeğe geçiş yapılmaktadır.

Grafik işlem birimleri (Graphics processing unit - GPU) çok çekirdekli (many core) bir mimariyi temsil etmekte ve daha önce yukarıda açıklanan Tek Komut Çoklu Veri ve Çoklu Komut Çoklu Veri paralelliğine sahiptir. NVIDIA, bu tür bir mimari için Tek Komut Çoklu Thread – TKÇT (Single Instruction Multiple Thread - SIMT) [9] sözcüğünü icat etmiştir. Her ne kadar ikisinde de çekirdek ifadesi kullanılsa da bir GPU çekirdeği CPU

çekirdeğinden oldukça farklıdır. Göreceli olarak ağır işler için olan bir CPU çekirdeği, çok karmaşık kontrol mantığı ve ardışık programların yürütülmesini optimize etmek için tasarlanmıştır. Göreceli olarak hafif işler için olan bir GPU çekirdeği, veri-paralel görevler için daha basit kontrol mantığı ile optimize edilmiş ve paralel programlarda yapılan işlem sayısının artırılmasına odaklanmıştır.

1.3. Heterojen Hesaplama

Eskiden, bilgisayarlar genel programlama görevlerini yerine getirmek üzere sadece merkezi işlemci birimlerini (CPU) kullanılmaktaydı. Zamanla, GPU'lar daha güçlü ve daha genel hale geldi ve yüksek performans ve güç verimliliği ile genel amaçlı paralel hesaplama görevlerini üstlenmeye başladılar. Homojen sistemlerden heterojen sistemlere geçiş, yüksek performanslı hesaplama geçmişi için bir kilometre taşıdır. Homojen hesaplama (Homogeneous computing), bir uygulamayı çalıştırmak için aynı mimarinin bir veya daha fazla işlemcisini kullanmaktadır [10]. Heterojen hesaplama (Heterogeneous computing) ise bir uygulamayı çalıştırmak için birden fazla işlemci mimarisi kullanmaktadır [11]. Heterojen hesaplama, görevleri uygun olan mimarilere atayarak sonuç olarak performans iyileştirmesi sağlamaktadır.

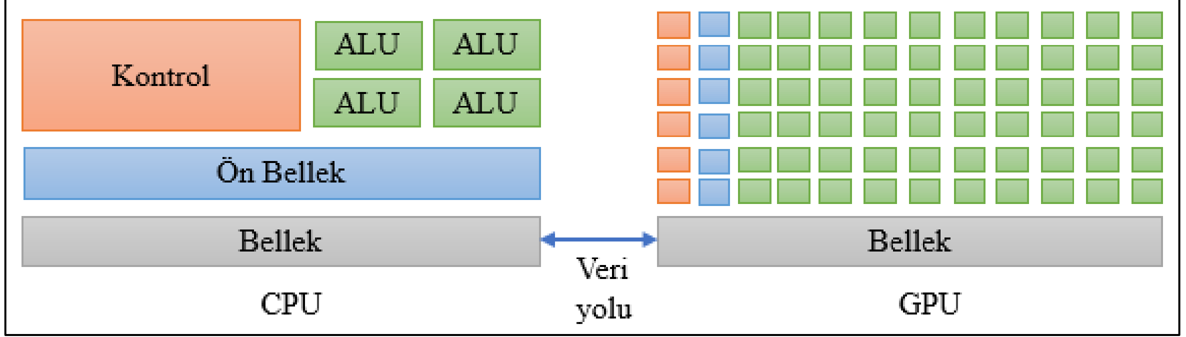
Heterojen sistemler geleneksel yüksek performanslı hesaplama sistemlerine kıyasla belirgin avantajlar sunmakla birlikte, artan uygulama tasarım karmaşıklığı nedeniyle bu tür sistemlerin etkin kullanımı sınırlı kalmaktadır.

1.3.1. Heterojen Mimari

Günümüzde tipik heterojen bir sistem, birkaç çoklu çekirdekli (multicore) CPU soketi ve iki veya daha fazla çok çekirdekli (many core) GPU'dan oluşmaktadır [12]. GPU şu anda bağımsız bir platform değildir. Fakat CPU'ya yardımcı işlemci konumunda bulunmaktadır. Bu nedenle GPU'lar Şekil 1.5.'deki gösterildiği gibi bir PCI-Express veri yolu üzerinden CPU tabanlı bir ana makine (host) ile birlikte çalışmaktadır. Bu nedenle, GPU hesaplama terimlerinde, CPU'ya ana bilgisayar (host), GPU'ya aygıt (device) denmektedir.

Şekil 1.5’in sağ tarafındaki küçük yeşil kutular GPU’daki Aritmetik ve Mantık Birimini (Arithmetic and Logic Unit – ALU), küçük mavi kutular ön belleği ve küçük

kırmızı kutular kontrol birimini göstermektedir. Şekil 1.5, çoklu çekirdekli (multicore) ve çok çekirdekli (many core) kavramını çok net bir şekilde göstermektedir.



Şekil 1.5. CPU ve GPU arasındaki mimari farkı

Heterojen bir uygulama iki bölümden oluşmaktadır:

- Ana bilgisayar (CPU) kodu
- Aygıt (GPU) kodu

Ana makine (ana bilgisayar) kodu CPU'larda ve aygıt kodu GPU'larda çalışmaktadır [13]. Heterojen bir platformda çalışan bir uygulama genellikle CPU tarafından başlatılmaktadır. CPU kodu, hesaplama yoğun görevleri aygıta yüklemeden önce aygıtını ortamını, kodunu ve verilerini yönetmekten sorumludur.

Hesaplama yoğun uygulamalarda, program bölümleri genellikle zengin bir veri paralelliği sergilemektedir. GPU'lar, bu bölümlerdeki veri paralelleştirilmesinin yürütülmesini hızlandırmak için kullanılmaktadır. Fiziksel olarak CPU'dan ayrı bir donanım bileşeni, bir uygulamanın hesaplama yoğun bölümlerini hızlandırmak için kullanıldığında, bir donanım hızlandırıcısı olarak anılmaktadır.

GPU kabiliyetini tanımlayan iki önemli özellik vardır:

- Çekirdek sayısı
- Bellek büyüklüğü

Buna göre, GPU performansını açıklayan iki farklı metrik vardır:

- Maksimum hesaplama performansı
- Bellek bant genişliği

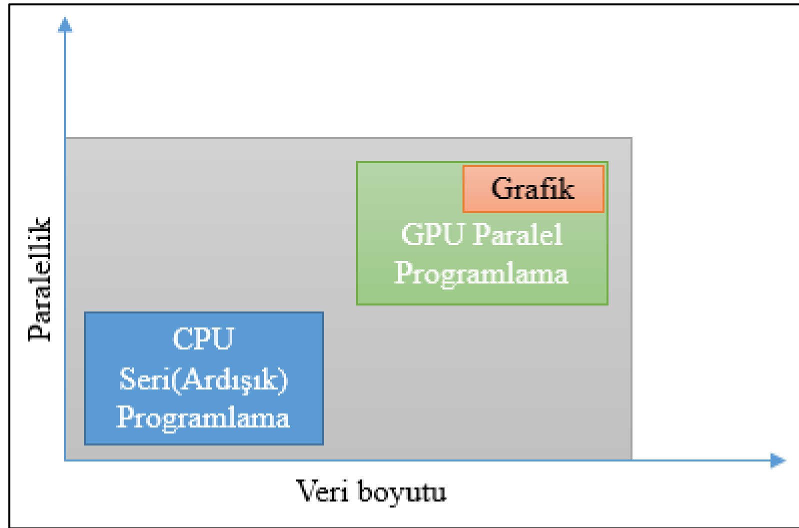
Maksimum hesaplama performansı (Peak computational performance), hesaplama kapasitesinin ölçümüdür ve genellikle saniyede kaç tane tek (float) veya çift (double)

duyarlılıklı kayan noktalı sayı hesaplayabileceği olarak tanımlanmaktadır. Maksimum performans genellikle Gflops (saniyede milyar kayan noktalı işlem) veya Tflops (saniyede trilyon kayan noktalı işlem) cinsinden ifade edilmektedir. Bellek bant genişliği (Memory bandwidth) verinin belleğe yazılma ya da bellekten okunma hızının bir ölçüsüdür. Bellek bant genişliği genellikle saniyede gigabayt (GB/s) cinsinden ifade edilmektedir.

1.3.2. Heterojen Hesaplama Paradigması

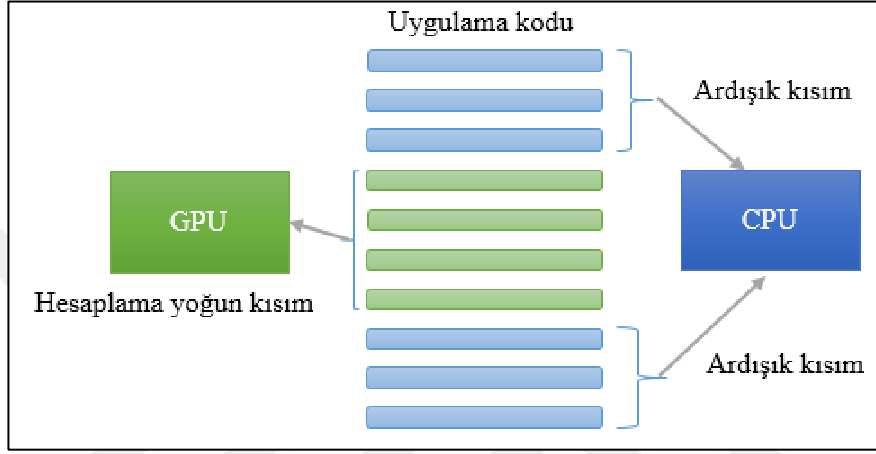
GPU hesaplamanın CPU hesaplamasının yerini alacağı söylenemez. Her yaklaşımın bazı program türleri için avantajları vardır. CPU hesaplama, kontrol yoğun görevler için uygundur. GPU hesaplama ise, veri paralel hesaplama yoğun görevler için uygundur. CPU'lar GPU'lar tarafından tamamlandığında, güçlü bir birleşim yaratmaktadır. CPU, öngörülemeyen kontrol akışı ve kısa hesaplama işlemler dizisi için optimize edilmiştir. GPU spektrumun diğer ucunu hedeflemektedir: basit kontrol akışı olan hesaplamalı görevlerin egemen olduğu iş yükleri için optimize edilmiştir. Şekil 1.6.'da gösterildiği gibi, CPU ve GPU için uygulama kapsamını ayıran iki boyut vardır:

- Paralellik seviyesi
- Veri büyüklüğü



Şekil 1.6. CPU ve GPU çalışma alanları

CPU + GPU heterojen paralel hesaplama mimarileri gelişmiştir, çünkü CPU ve GPU, uygulamanın her iki tür işlemci kullanarak en iyi performansı almasını sağlayan tamamlayıcı niteliklere sahiptir. Bu nedenle, uygulamalarda en iyi performans için CPU ve GPU'nun birlikte kullanılması gerekebilir. Şekil 1.7.'de gösterildiği gibi, CPU üzerinde sıralı parçaları veya görev paralel kısımları ve GPU üzerinde ise yoğun veri paralel kısımları çalıştırmak gerekebilir.



Şekil 1.7. CPU ve GPU'nun çalıştırdığı kod kısımları

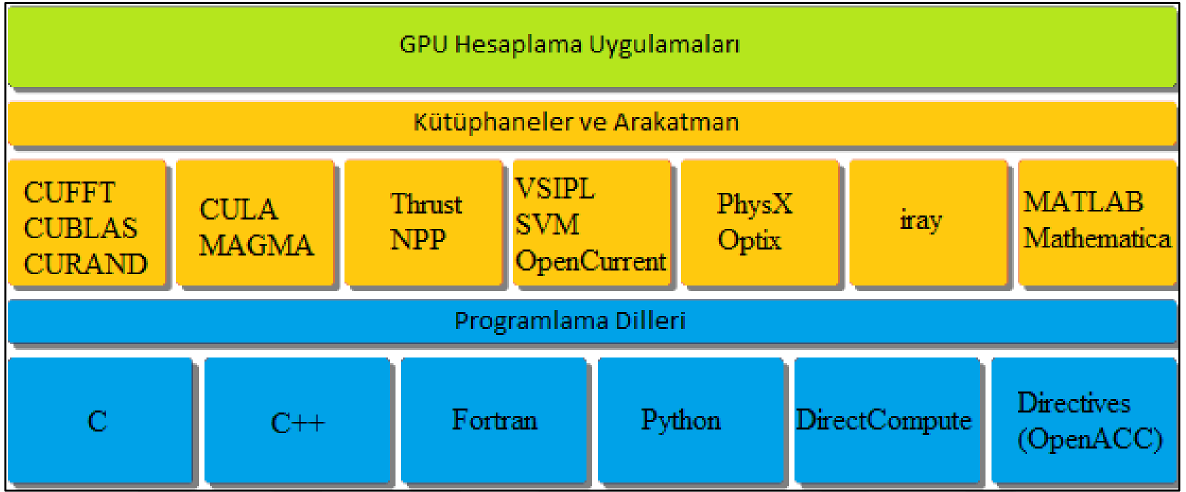
Bu şekilde kod yazma, GPU ve CPU'nun özelliklerinin birbirini tamamladığından emin olunmasını sağlamaktadır. CPU ve GPU'nun birleştirilmesi sayesinde sisteminin hesaplama gücünden tam olarak yararlanabilmektedir. NVIDIA, bir uygulamanın ortak CPU+GPU yürütmesini desteklemek için CUDA adlı paralel hesaplama mimarisi tasarlamıştır.

1.3.3. CUDA: Heterojen Hesaplama İçin Bir Mimari

CUDA, çok karmaşık hesaplama problemlerini daha verimli bir şekilde çözmek için NVIDIA GPU'lardaki paralel hesaplama altyapısını kullanan genel amaçlı bir paralel hesaplama mimarisidir [14]. CPU üzerinde geleneksel olarak yapıldığı gibi, CUDA kullanarak GPU'ya hesaplama için erişilebilmektedir.

CUDA mimarisine Şekil 1.8.'deki gibi, CUDA hızlandırılmış kütüphaneler, derleyici komutları, uygulama programlama arabirimleri (API) ve C, C ++, Fortran ve Python gibi

programlama dilleriyle erişilebilmektedir. CUDA C [15], heterojen programlamayı etkinleştirmek için bir dizi yeni özelliklerle birlikte standart ANSI C'nin bir uzantısıdır ve cihazları, hafızayı ve diğer görevleri yönetmek için basit bir uygulama programlama arayüzü (application programming interface - API) içermektedir. CUDA paralel hesaplama mimarisi ayrıca, programların, paralelliklerini farklı sayıda çekirdeği olan GPU'lara şeffaf bir şekilde ölçeklenmesine olanak tanıyan ölçeklenebilir bir programlama modeli sunmaktadır. C programlama dili ile aşina olan programcılar için hızlı bir öğrenme olanağı sağlamaktadır.



Şekil 1.8. CUDA API ve programlama dilleri

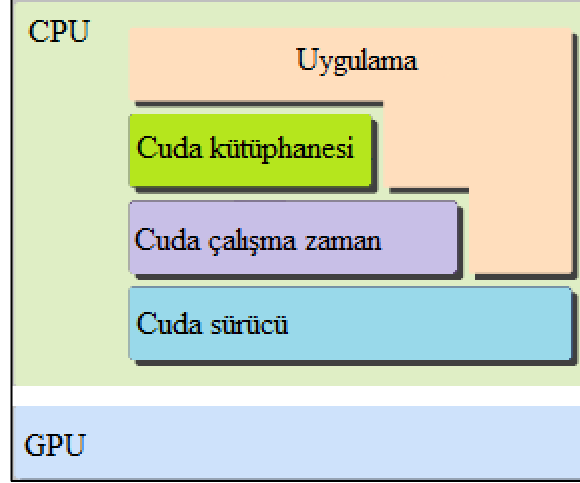
CUDA, Şekil 1.9.'da gösterildiği gibi GPU aygıtını yönetmek ve thread'leri düzenlemek için iki API düzeyi sağlamaktadır:

- CUDA Sürücü (Driver) API
- CUDA Çalışma Zamanı (Runtime) API

Driver API düşük seviyeli bir API'dir ve programlanması nispeten zordur ancak GPU cihazının nasıl kullanıldığı üzerinde daha fazla kontrol sağlamaktadır [16]. Runtime API, driver API'sinin üst kısmında uygulanan daha üst düzey bir API'dir [17]. Runtime API'sinin her bir işlevi, driver API'sine verilen daha temel işlemlere ayrılmıştır.

Bir CUDA programı aşağıdaki gibi iki bölümden oluşan bir karışımdan oluşur:

- CPU üzerinde koşan ana makine (bilgisayar) kodu
- GPU üzerinde koşan cihaz (aygıt) kodu

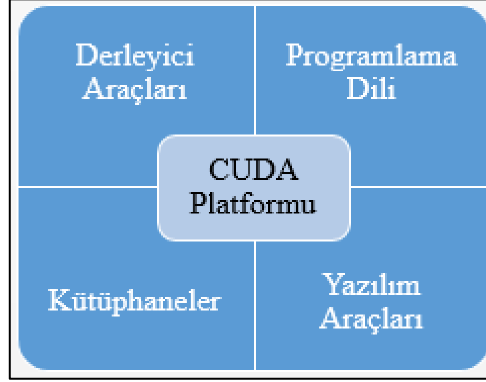


Şekil 1.9. CUDA API seviyesi

NVIDIA'nın CUDA nvcc derleyicisi, derleme işlemi sırasında cihaz kodunu ana makine kodundan ayırmaktadır [18]. Ana makine kodu standart C kodu olup ayrıca C derleyicileri ile derlenmektedir. Cihaz kodu, veri-paralel işlevlerin etiketlenmesi için anahtar sözcüklerle genişletilmiş CUDA C kullanılarak yazılmaktadır; buna çekirdek (kernel) denir [19]. Cihaz kodu ayrıca nvcc tarafından derlenmektedir. Bağlantı aşamasında (link stage) çekirdek fonksiyon çağrıları için CUDA Runtime kütüphaneleri eklenmektedir. CUDA nvcc derleyicisi, yaygın olarak kullanılan LLVM açık kaynak derleyici altyapısına dayanmaktadır. CUDA Derleyici yazılım geliştirme aracını (software development kit-SDK) kullanarak GPU hızlandırması desteğiyle programlama dilleri oluşturabilmekte veya genişletebilmektedir.

CUDA paralel hesaplama platformu, Şekil 1.10.'da gösterildiği gibi, çeşitli paralel hesaplama ekosistemini destekleyen bir temel oluşturmaktadır.

Bugün, giderek artan sayıda şirket dünya standartlarında araçlar, hizmetler ve çözümler sunarak CUDA ekosistemini hızla büyütmektedir. Uygulamalar GPU'larda oluşturulmak istenirse, CUDA Araç Seti'yle (CUDA Toolkit) GPU'ların performansı en kolay şekilde kullanabilmektedir [20]. CUDA Toolkit C ve C++ geliştiricileri için kapsamlı bir geliştirme ortamı sağlamaktadır. CUDA Toolkit, uygulamaların performansını artırmak ve hata ayıklamak (debugging) için bir derleyici, matematik kütüphanesi ve araçlar (tools) içermektedir. CUDA Toolkit ayrıca, kod örnekleri, programlama kılavuzları, kullanım kılavuzları, API referansları ve yardımcı olacak diğer belgeleri içermektedir.



Şekil 1.10. CUDA paralel hesaplama platformu

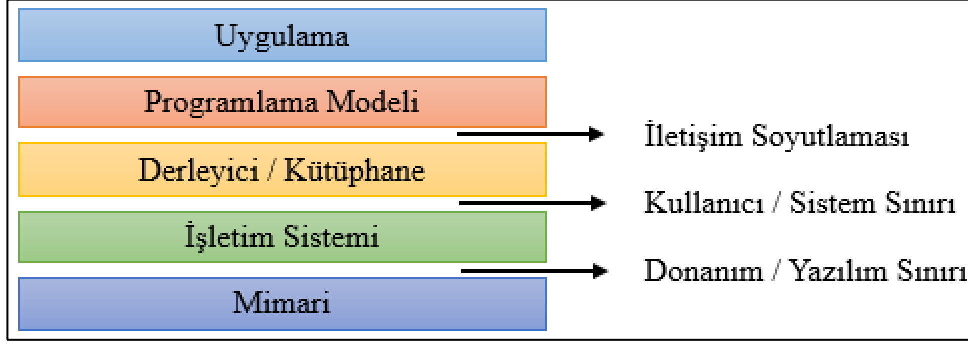
1.4. CUDA

CUDA, C dilinde küçük birkaç eklenti ile oluşturulmuş paralel hesaplama mimarisidir. CUDA ile paralel bir algoritma, C programı yazıyormuş gibi kolayca uygulanabilmektedir. NVIDIA GPU'ları ile gömülü cihazlarda, tabletlerde, dizüstü bilgisayarlarında, masaüstü bilgisayarlarında ve iş istasyonlarından yüksek performanslı hesaplama kümelerine kadar geniş bir yelpazede sayısız CUDA sistemi oluşturulabilmektedir. C programlama yazılımı araçlarına benzer araçlar ile proje boyunca CUDA programı düzenlenebilmekte, hata ayıklanabilmekte (debug) ve analiz edilebilmektedir.

1.4.1. CUDA Programlama Modeli

Programlama modelleri, bilgisayar mimarilerinin soyutlamasını sağlamaktadır. Ayrıca bir uygulama ile o uygulamanın mevcut donanım üzerinde yürütülmesi arasında köprü görevi görmektedir. Program ile programlama modeli uygulaması arasındaki soyutlamanın önemli katmanları Şekil 1.11.'deki gibidir.

İletişim soyutlaması, program ile programlama modeli uygulaması arasındaki sınırdır. İletişim soyutlaması, ayrıcalıklı donanım ve işletim sistemi özellikleri kullanılarak derleyici veya kütüphaneler tarafından gerçekleştirilmektedir. Programlama modeli için yazılan program, program bileşenlerinin bilgi paylaşımını ve faaliyetlerini koordine etmesini belirlemektedir. Programlama modeli, özel hesaplama mimarilerine mantıksal bir bakış açısı sağlamaktadır. Programlama modeli genellikle, bir programlama dili veya programlama ortamında somutlaşmaktadır.



Şekil 1.11. Program ve program modelinin uygulaması arasındaki soyutlama

Diğer paralel programlama modellerinde olan birçok soyutlama özelliğine sahip olmasının yanında, CUDA programlama modeli, GPU mimarilerinin hesaplama gücünü kullanmak için aşağıdaki iki önemli özelliği de sahiptir:

- Bir hiyerarşi yapısı aracılığıyla GPU'daki thread'leri düzenlemek
- Bir hiyerarşi yapısı vasıtasıyla GPU'daki belleğe erişmek

Bir programcının perspektifinden, paralel hesaplama farklı seviyelerden incelenebilir[4]:

- Domain seviyesi
- Mantık seviyesi
- Donanım seviyesi

Program ve algoritma tasarımı boyunca çalışırken ana kaygı domain düzeyindedir. Paralel bir ortamda çalışırken sorunu doğru ve verimli bir şekilde çözmek için veriler ve işlevler nasıl ayrıştırılır cevabı domain seviyesinde aranmaktadır. Programlama aşamasına girildiğinde endişe, eşzamanlı thread'lerin nasıl düzenleneceğine kaymaktadır. Bu aşamada, thread'lerin ve hesaplamaların sorunu doğru bir şekilde çözdüğünden emin olmak için mantık seviyesinde düşünülmesi gerekmektedir. C paralel programlamada, thread'ler pthread veya OpenMP teknikleri kullanılarak açıkça yönetilmelidir. CUDA, thread'leri kontrol etmek için hiyerarşik bir thread soyutlaması sunmaktadır. Bu soyutlama, paralel programlama için üstün ölçeklenebilirlik sunmaktadır. Donanım düzeyinde, thread'lerin çekirdeklerle (core) nasıl eşlendiğini anlayabilmek, performansı artırmaya yardımcı olmaktadır. CUDA thread modeli, çok fazla alt seviye ayrıntıya girmeden size yeterli bilgi sunmaktadır.

1.4.1.1. CUDA Programlama Yapısı

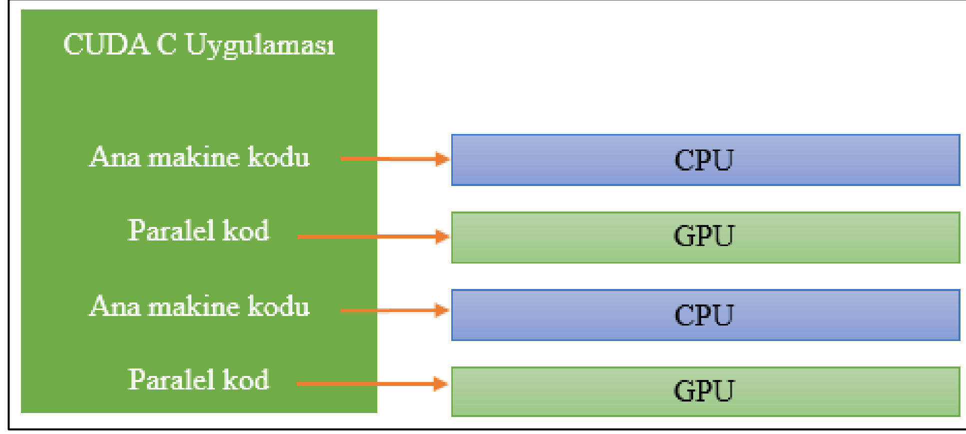
CUDA programlama modeli, C diline çok benzer CUDA C ile heterojen hesaplama sistemleri üzerinde uygulamaların yürütülmesini sağlamaktadır. Heterojen bir ortam, her biri kendi bellek adresine sahip ve PCI-Express veri yolu ile ayrılmış GPU'lar tarafından tamamlanan CPU'lardan oluşmaktadır. Bu nedenle, aşağıdaki ayrıma dikkat etmek gerekmektedir:

- Host (Ana bilgisayar): CPU ve kendi belleği
- Device (Aygıt): GPU ve kendi belleği

Programcı tarafından yönetilen bellek ve verilerin kontrolü, uygulamayı optimize etme ve donanım kullanımını en üst düzeye çıkarma olanağı vermektedir.

CUDA programlama modelinin önemli bir bileşeni GPU aygıtında çalışan koddur. Bu koda kernel (çekirdek) denmektedir [19]. Geliştirici açısından, bir çekirdek ardışık bir program olarak ifade edilebilir. Sahnenin arkasında CUDA, GPU thread'leri üzerinde programlayıcı tarafından yazılmış kernel'in zamanlamasını yönetmektedir. Host tarafından ise algoritmanın uygulama verisine ve GPU özelliğine göre cihaza (device) nasıl uygulanacağı tanımlanmaktadır. Amaç, algoritmanın mantığına basit bir şekilde (ardışık kod yazarak) odaklanılmasını ve binlerce GPU thread oluşturma ve yönetme ayrıntılarıyla boğuşulmamasını sağlamaktır.

Host (CPU), çoğu işlem için device (GPU) bağımsız olarak çalışabilmektedir. Bir kernel başlatıldığında, kontrol anında host'a geri döndürülmekte ve CPU'yu device'da çalışan veri paralel kodun ilave görevlerini gerçekleştirmek için serbest bırakmaktadır. Tipik bir CUDA programı, Şekil 1.12.'de gösterildiği gibi paralel kod ile tamamlanan seri koddan oluşmaktadır. Paralel kod GPU cihazında yürütülürken, seri kod (ve komut paralel kodu) host'da yürütülmektedir. Host kodu ANSI C'de yazılmıştır ve device kodu CUDA C kullanılarak yazılmıştır. Tüm kod tek bir kaynak dosyasına koyulabilmektedir veya uygulamalar veya kitaplıklar oluşturmak için birden fazla kaynak dosya kullanılabilmektedir. NVIDIA C Derleyici (nvcc) hem host hem de device için yürütülebilir kodu üretmektedir.



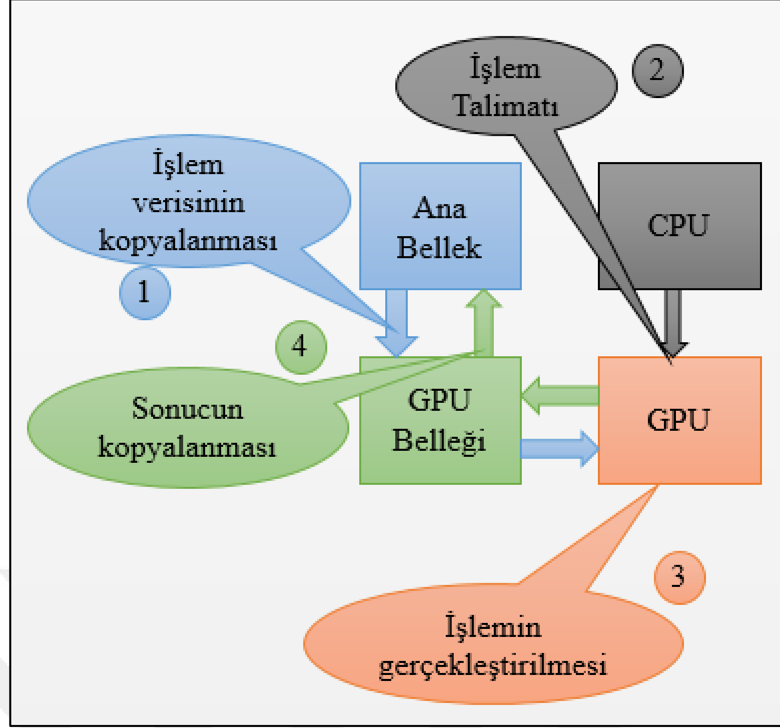
Şekil 1.12. Tipik bir CUDA uygulaması

Bir CUDA programının tipik bir iş akışı Şekil 1.13.'de gösterildiği gibi 4 aşamadan oluşmaktadır:

1. CPU hafızasından GPU hafızasına veri kopyalama
2. GPU belleğinde saklanan veriler üzerinde çalışmak için kernel çağrılması
3. GPU üzerinde verilerin işlenmesi
4. GPU belleğinden verileri CPU belleğine geri kopyalama

1.4.1.2. Thread Organizasyonu

Host tarafında bir kernel başlatıldığında yürütme device'a taşınmaktadır. Burada çok sayıda thread üretilmekte ve her thread çekirdek işlevi tarafından belirtilen ifadeleri yürütmektedir. Thread'lerin nasıl düzenleneceğini bilmek CUDA ile programlamanın kritik bir parçasıdır. CUDA paralel hesaplama mimarisi, thread düzenlemesini sağlamak için bir thread hiyerarşi soyutlaması sunmaktadır. Bu hiyerarşi, Şekil 1.14.'de gösterildiği gibi, blok içerisindeki thread'ler ve grid içerisindeki bloklar olmak üzere iki seviyeli bir hiyerarşidir [21]. Grid içinde bloklar ve blok içinde thread'ler 1, 2 veya 3 boyutlu olarak dizilebilmektedir. Örneğin Şekil 1.14.'deki grid içerisindeki blok hiyerarşisi 2 boyutludur (x = 2 ve y = 2) . Aynı şekil için blok içerisindeki thread hiyerarşisi de 2 boyutludur (x = 3 ve y = 2).

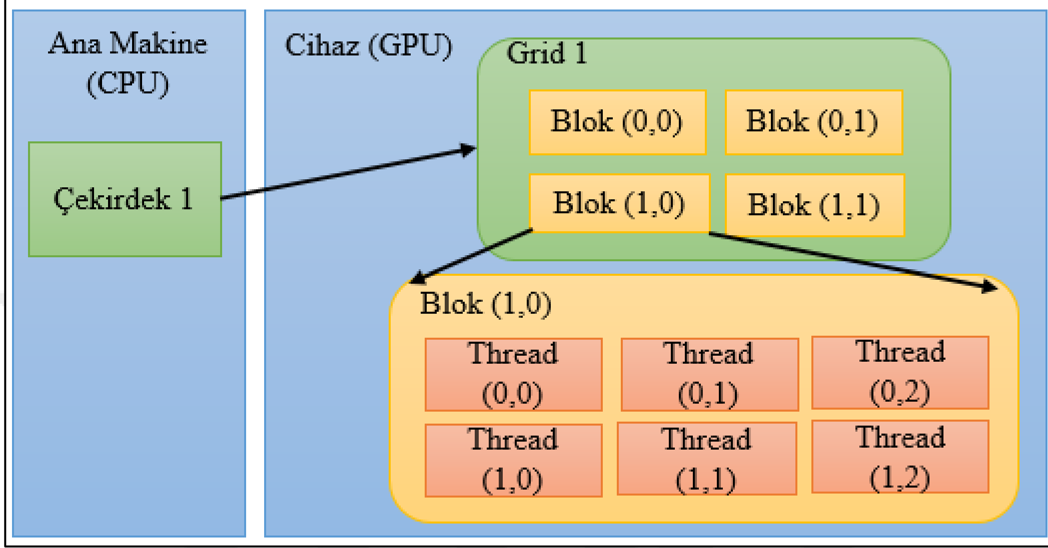


Şekil 1.13. CUDA iş akışı

Tek bir kernel'in tarafından üretilen tüm thread'ler topluca bir grid olarak adlandırılmaktadır. Bir grid içindeki tüm thread'ler aynı bellek alanını paylaşmaktadır. Grid birçok thread bloğundan oluşmaktadır. Bir thread bloğu, birbiriyle işbirliği yapabilen bir grup thread'ten oluşmaktadır. Farklı bloklardaki thread'lerin birbirleriyle iş birliği mümkün değildir. Grid ve blok yapısı bir, iki ve üç boyutlu olabilmektedir. Grid ve blok sayısının en önemli kısıtlayıcı faktörü bellek, kaydediciler (register) gibi GPU kaynaklarıdır. GPU kaynakları blok ve thread'ler arasında bölüşüldüğü için eğer yeteri kaynak yoksa grid ve blok sayısının küçültülmesi gerekebilir. Grid ve blok yapısı farklı olması performansı etkilemektedir. Bunun nedeni, üzerinde çalıştığı mimaridir. CUDA içerisinde, 32 thread'ten oluşan yapılara warp (çözgü) denir [22]. Warp en küçük işlem birimidir. Eğer 33 thread çalıştırılması gereken bir problem varsa, en küçük işlem birimi warp olduğundan, CUDA 2 warp'lık kaynak ayırmaktadır. Bu yüzden 32 sayısı CUDA için çok önemli bir sayıdır. Eğer grid ve blok boyutları 32'in katları olursa program daha performanslı çalışacaktır.

Bir C programcısı, uygulamanın doğru bir şekilde çalışması için kod yazarken, önbellek özelliklerini yok sayılabilmektedir. Ancak, performansı için kod yazarken kod yapısındaki önbellek özellikleri göz önüne almak gerekmektedir. Bu durum, CUDA C

programlaması için de geçerlidir. CUDA C programcısı olarak kernel performansı iyileştirilmek istenirse donanım kaynaklarını bir miktar anlamak gerekmektedir. Donanım mimarisini anlaşılmazsa, CUDA derleyicisi kernel için yine de optimizasyon yapmaya çalışacaktır, ancak herhangi bir garantisi yoktur.



Şekil 1.14. CUDA thread hiyerarşisi

1.4.2. CUDA Bellek Modeli

Bir önceki bölümde, farklı grid ve blok organizasyonun performansı etkilediği belirtilmişti. Ama performansı etkileyen tek faktör thread organizasyonu değildir. Bu thread'lerin belleğe yazma ve bellekten okuma verimleri de performansı büyük ölçüde etkilemektedir. Bellek erişimi ve yönetimi, herhangi bir programlama dilinin önemli bir parçasıdır. Bellek yönetimi, modern yüksek performanslı sistemler üzerinde özellikle büyük bir etkiye sahiptir. Birçok iş yükü, verileri ne kadar hızlı bir şekilde yazıp okuyabildiği ile sınırlıdır. Çünkü düşük gecikmeli ve yüksek bant genişliğinde bir belleğe sahip olmak performansa çok yararlı olabilir. Fakat büyük kapasiteli ve yüksek performanslı belleğin sağlanması her zaman mümkün veya ekonomik değildir. Bunu yerine var olan bellek üzerinde minimum gecikme ve maksimum veri yolu genişliği elde etmek için bir bellek modelinin kullanılması gerekmektedir. CUDA bellek modeli, host ve device bellek sistemlerini birleştirmekte ve en iyi performans için veri yerleşimini açık şekilde kontrol edebilmek adına tam bellek hiyerarşisini sunmaktadır [23].

1.4.2.1. Bellek Hiyerarşisinin Faydaları

Genel olarak, uygulamalar herhangi bir zamanda herhangi bir veriye erişmez veya rasgele kod çalıştırmamaktadırlar. Bunun yerine, uygulamalar genellikle yerellik (locality) ilkesine uymaktadırlar [24]. Bu, herhangi bir zamanda adres alanlarının nispeten küçük ve yerel bir bölümüne eriştiklerini göstermektedir. İki farklı yerellik türü vardır:

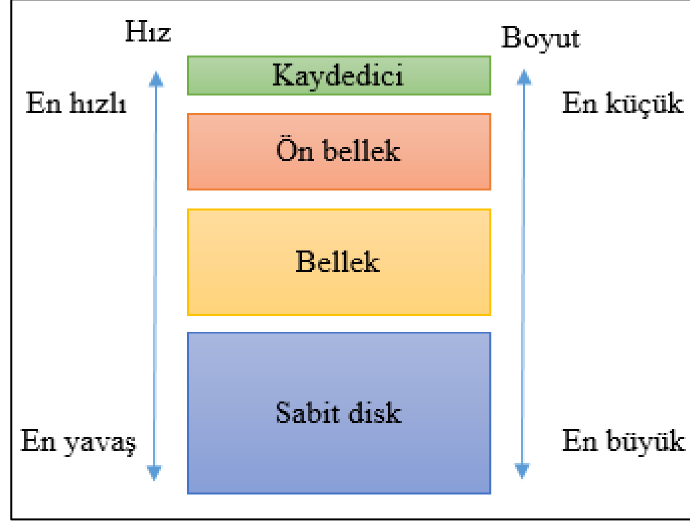
- Zamansal yerellik
- Uzaysal yerellik

Zamansal yerellik, bir veri konumuna erişilirse, kısa bir zaman aralığında yeniden erişilmesi olasıdır ve zaman geçtikçe erişilme olasılığının düşük olduğu varsayılmaktadır. Uzaysal yerellik, bir bellek konumuna erişilirse, yakındaki yerlerin de erişilmesinin muhtemel olduğunu varsaymaktadır.

Modern bilgisayarlar, performansı en iyi duruma getirmek için kademeli olarak daha düşük gecikmeli fakat daha düşük kapasiteli bir bellek hiyerarşisi kullanmaktadır. Bu bellek hiyerarşisi, yalnızca yerellik ilkesinden dolayı kullanışlıdır. Genel olarak, bellek kapasitesi arttıkça, işlemci-bellek gecikmesi artmaktadır. Tipik bir hiyerarşi Şekil 1.15.'deki gibidir. Şekil 1.15.'in alt kısmındaki depolama türleri genel olarak şu özelliklerle karakterize edilmektedir:

- Bit başına daha düşük maliyet
- Yüksek kapasite
- Yüksek gecikme
- İşlemci tarafından daha az erişilme

Hem CPU'lar hem de GPU'lar için ana bellek, DRAM (Dinamik Rasgele Erişim Belleği) kullanılarak gerçekleştirmektedir. Daha düşük gecikmeli bellek (CPU L1 önbellek gibi) SRAM (Statik Rasgele Erişim Belleği) kullanılarak gerçekleştirilmektedir. Bellek hiyerarşisinde en büyük ve en yavaş seviye genellikle bir manyetik disk veya flash sürücü kullanılarak gerçekleştirilmektedir. Bu bellek hiyerarşisinde, veriler işlemci tarafından aktif olarak kullanıldığında düşük düzeyli, düşük kapasiteli bellek veya daha sonra kullanmak üzere depolandığı zaman yüksek gecikmeli, yüksek kapasiteli bellekte tutulmaktadır. Bu bellek hiyerarşisi, büyük ama gecikme süresi düşük bellek yanılsamasını sağlamaktadır.



Şekil 1.15. Bellek hiyerarşisi

Hem GPU'lar hem de CPU'lar, bellek hiyerarşi tasarımında benzer prensipleri ve modelleri kullanmaktadır. GPU ve CPU bellek modelleri arasındaki en önemli fark, CUDA programlama modelinin bellek hiyerarşisini daha açık bir şekilde ortaya koyması ve programcılara davranış üzerinde daha açık bir kontrol hakkı vermesidir.

1.4.2.2 Bellek Modelinin Bileşenleri

Programcılara göre hafıza genellikle ikiye ayrılmaktadır:

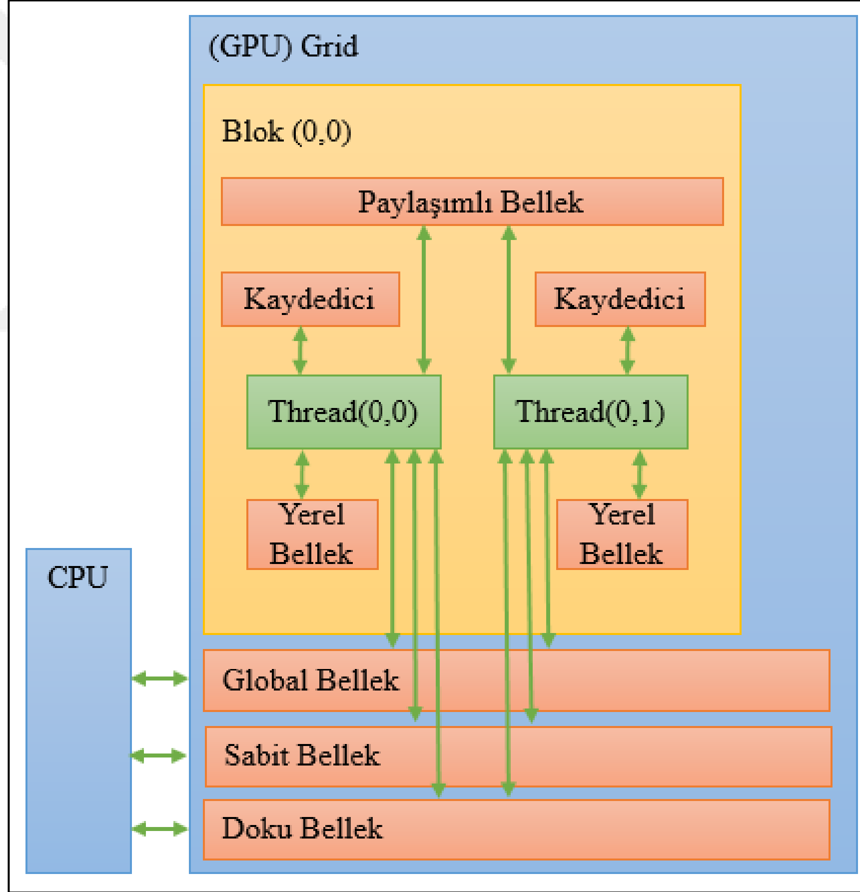
- Programlanabilir: Hangi verilerin programlanabilir belleğe yerleştirildiği açıkça kontrol edilmektedir.
- Programlanamaz: Veri yerleşimi üzerinde hiçbir kontrole sahip olunmaz ve iyi performans elde etmek için otomatik teknikler kullanılmaktadır.

CPU bellek hiyerarşisinde L1 ve L2 önbellek programlanabilir olmayan bellek örnekleridir. Öte yandan, CUDA bellek modeli birçok programlanabilir belleği programcıya sunmaktadır:

- Kaydedici (Registers)
- Paylaşımlı bellek (Shared memory)
- Yerel bellek (Local memory)
- Sabit bellek (Constant memory)
- Doku bellek (Texture memory)

- Global bellek (Global memory)

Şekil 1.16. bu bellek alanlarının hiyerarşisini göstermektedir. Her birinin kapsamı, ömrü ve önbellekleme davranışları farklıdır. Çekirdeğin (kernel) içindeki thread kendi özel (private) yerel belleğine sahiptir. Bir thread bloğu, aynı thread bloğundaki tüm thread'ler tarafından görülebilen ve içeriği thread bloğu ömrü boyunca devam eden kendi paylaşımlı belleğine sahiptir. Bütün thread'ler global belleğe erişebilmektedir. Ayrıca, tüm thread'ler tarafından erişilebilen iki salt okunur bellek alanı vardır: sabit ve doku bellek alanları. Global, sabit ve doku bellek alanları farklı kullanımlar için optimize edilmiştir. Doku belleği çeşitli veri düzenleri için farklı adres düzeni ve filtreleme sunmaktadır. Global, sabit ve doku belleği içeriği, bir uygulama ile aynı yaşam süresine sahiptir.



Şekil 1.16. CUDA bellek modeli

Kaydediciler bir GPU'nun en hızlı bellek alanını oluşturmaktadır. Bir çekirdekte başka bir tür niteleyici olmadan bildirilen otomatik bir değişken genellikle bir kaydedicide

saklanmaktadır. Kaydedici değişkenleri her thread'e özeldir. Bir çekirdek (kernel), sık erişim sağlanan thread'e özel değişkenlerini depolamak için kaydedicileri kullanmaktadır. Kaydedicilerin ömrü çekirdek ile aynıdır. Çekirdek çalışmayı sonlandırdığında kaydedicilere ulaşım sağlanmaz. Kaydediciler nadir kaynaklardır. Akışlı çoklu işlemci (streaming multiprocessor - SM) içindeki aktif warp'lar arasında dağıtılmaktadırlar. Çekirdeğinizde az kaydedici kullanılması daha fazla blok ve thread kullanılmasına olanak sağlamaktadır. Bu durum, donanımdan maksimum oranda faydalanılmasını ve performansın artırılmasını sağlamaktadır. Eğer bir çekirdek (kernel) kaydedici kullanımında donanım limitlerini aşarsa yerel bellek kullanılmaya başlanacaktır. Bu durumda, performans olumsuz bir şekilde etkilenecektir. O yüzden bu nadir kaynaklar çok dikkatli bir şekilde kullanılmalıdır.

Yerel bellek, çekirdekte kaydedici sayısının fazla olduğu zamanlarda kullanılan bellektir. "Yerel bellek" ismi yanıltıcı olabilir: Yerel belleğe gelen değerler, global bellek ile aynı fiziksel konumda bulunmaktadır, bu nedenle yerel bellek erişimi yüksek gecikme ve düşük bant genişliği ile karakterize edilmektedir.

Paylaşımlı bellek, CPU L1 önbelleğine benzer bir şekilde kullanılmaktadır, ancak aynı zamanda programlanabilirdir [25]. Paylaşımlı bellek yonga üzerinde olduğundan, yerel veya global bellekten daha yüksek bir bant genişliği ve çok daha düşük gecikme süresi vardır. Her bir SM, thread blokları arasında bölünen sınırlı miktarda paylaşılan bellek içermektedir. Bu nedenle, paylaşılan hafıza aşırı derecede kullanılmamaya özen gösterilmelidir yoksa aktif warp sayısı istemeden sınırlanmaktadır. Paylaşımlı bellek, bir çekirdek işlevi kapsamında bildirilir ancak ömrünü bir thread bloğuyla paylaşır. Bir thread bloğu yürütülürken tamamlandığında, paylaşımlı belleğin tahsisi serbest bırakılmaktadır ve diğer thread bloklarına atanmaktadır.

Sabit bellek, cihaz belleğinde ve her SM için ona ayrılmış bir bellek adresinde bulunmaktadır. Sabit değişkenler, tüm çekirdeklerin dışında global kapsamda (scope) bildirilmelidir. Sınırlı miktarda sabit bellek bildirilebilmektedir. Sabit bellek, statik olarak bildirilmekte ve aynı derleme birimindeki tüm kernel'lara görünür durumdadır. Sabit bellek, bir warp içindeki tüm thread'ler aynı bellek adresinden okunduğunda en iyi sonucu vermektedir. Örneğin, bir matematiksel formül için bir katsayı, sabit belleğe iyi bir kullanım örneğidir. Çünkü bir warp içindeki tüm thread'ler, aynı hesaplamayı farklı veriler üzerinde yapmak için aynı katsayıyı kullanmaktadır. Bir warp'daki her thread farklı bir adresten okur

ve yalnızca bir kez okursa, sabit bellek en iyi seçim değildir. Çünkü sabit bellekteki tek bir okuma, warp içindeki tüm thread'lere yayınlanmaktadır (broadcast) .

Doku bellek, cihaz belleğinde bulunur ve SM için ona ayrılmış bir bellek adresinde bulunur. Salt okunur (read-only) bir bellektir. Doku belleği, salt okunur bir önbellek aracılığıyla erişilen global bellek türüdür. Salt okunur önbellek, salt okunur işlemin bir parçası olarak kayan nokta interpolasyonunu (ara değerleme) gerçekleştirebilen donanım filtreleme desteğini içermektedir. Doku bellek 2 boyutlu uzaysal yerellik için optimize edilmiştir, bu nedenle 2 boyutlu verilerine erişmek için doku belleği kullanan warp içindeki thread'ler en iyi performansı elde edecektir. Bazı uygulamalar için bu idealdir ve önbellek ve filtreleme donanımı nedeniyle bir performans avantajı sağlamaktadır. Bununla birlikte, doku belleği kullanan diğer uygulamalar için global bellekten daha yavaş olabilmektedir.

Global bellek, bir GPU'nun en büyük, en yüksek gecikmeli ve en çok kullanılan bellek türüdür [26]. Global adı kapsamı ve ömrü anlamına gelmektedir. Durumuna, uygulamanın ömrü boyunca SM erişilebilir. Global bellek tahsisleri, bir uygulamanın ömrü boyunca var olmakta ve çekirdeklerdeki tüm thread'leri tarafından erişilebilmektedir. Birden çok thread global belleğe erişirken dikkatli olunmalıdır. Thread yürütme thread blokları arasında eşitlenemediğinden, eşzamanlı olarak global bellekte aynı konumu değiştiren, farklı thread bloklarındaki birden çok thread potansiyel bir tehlike oluşturmaktadır; bu, tanımlanmamış bir program davranışına neden olmaktadır. En iyi performansı elde etmek için bellek işlemlerini optimize etmek hayati önem taşımaktadır. Warp, bir bellek okuması / yazması yaptığında, bu talebi karşılamak için gereken işlemlerin sayısı genellikle aşağıdaki iki faktöre bağlıdır:

- Warp içindeki thread'lerin bellek adreslerinin dağılımı
- İşlem başına hafıza adreslerinin hizalanması

Genel olarak, bir bellek isteğini karşılamak için gereken daha fazla işlem oluyorsa, veri transferinin verimliliğinde bir azalmaya neden olmaktadır.

1.4.3. CUDA Akışlar (Streams)

Bir CUDA stream'i, bir cihaz (GPU) üzerinde, ana bilgisayar (CPU) kodu tarafından verilen sırayla çalıştırılan asenkronize CUDA işlemlerinin bir dizisini ifade etmektedir [27]. Stream özelliği, işlemlerin sırasının korunmasına, onların kuyruğa alınmasına ve kuyruğa alınmış işlemlerinin durumunun sorgulanmasına olanak sağlamaktadır. Bu işlemler, ana

bilgisayar (CPU) – cihaz (GPU) veri aktarımı, çekirdek (kernel) başlatmaları ve ana bilgisayar tarafından verilen fakat GPU tarafından kullanılan diğer komutların çoğunu içermektedir. Bu işlemler CPU ile asenkron olarak çalışmaktadır [28]. Yani bu işlemler başladıktan sonra kontrol tekrar CPU'ya dönmektedir. GPU'da yapılan işlerin bitmesi beklenmemektedir. CUDA çalışma zamanı (runtime) işlemlerin ne zaman GPU üzerinde çalışmak için hazır olacağını belirlemektedir. Sonuçları kullanmadan önce asenkron işlemin tamamlandığından emin olmak için CUDA API'leri kullanmak programcının sorumluluğundadır.

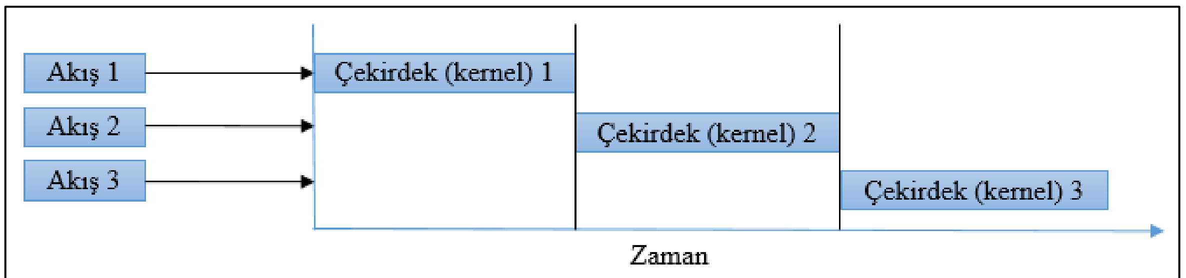
Genel olarak, CUDA C programlamasında iki eşzamanlılık düzeyi vardır:

- Çekirdek (kernel) seviyesinde eşzamanlılık
- Grid seviyesinde eşzamanlılık

Çekirdek seviyesindeki eşzamanlılıkta, tek bir görev veya çekirdek (kernel) GPU üzerindeki birçok thread tarafından paralel olarak yürütülmektedir. Buradaki eşzamanlılık, çekirdek (kernel) içindeki thread'lerin eşzamanlılığıdır. Bir thread bloğu içindeki thread'lerin eşzamanlı olarak yürütülebilmektedir. Fakat bloklar arası bir eşzamanlılık yoktur.

Grid seviyesindeki eşzamanlılıkta, GPU üzerinde birden fazla görev veya çekirdek (kernel) eşzamanlı olarak başlatılabilmektedir. Buradaki eşzamanlılık, GPU içindeki çekirdeklerin (kernel) eşzamanlılığıdır. Bu genellikle GPU'dan daha fazla yararlanılmasını sağlamaktadır.

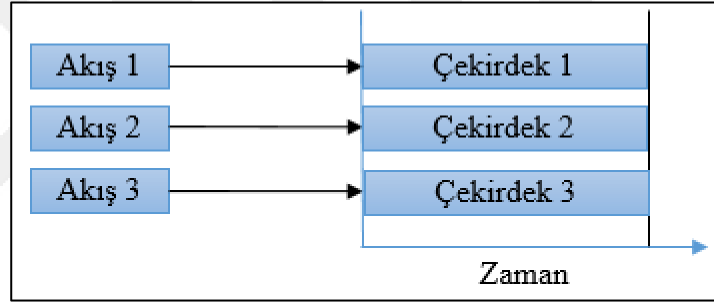
Şekil 1.17, grid seviyesinde eşzamanlılık olmadığındaki durumu göstermektedir. GPU üzerinde birden fazla çekirdek veya görev başlatılsa bile eğer grid seviyesinde eşzamanlılık yoksa cihaz (GPU) bunları Şekil 1.17'deki gibi sıralı bir şekilde çalıştıracaktır. Her bir akış bir çekirdeğin çalıştırılmasından sorumludur. Bu sorumluluğa, çekirdeğin çalıştırılmasından önceki ve çekirdek çalıştırıldıktan sonraki işlemler de dâhildir.



Şekil 1.17. Seri (eşzamanlı olmayan) akışlar

Aynı CUDA akışındaki işlemler sıkı bir sıralamaya sahipken, farklı akışlardaki işlemler yürütme sırası üzerinde herhangi bir kısıtlamaya sahip değildir. Birden fazla eşzamanlı çekirdek başlatmak için birden çok akış kullanılabilmekte ve grid seviyesinde eşzamanlılık elde edilebilmektedir. Bir CUDA akışında sıralanan tüm işlemler asenkron olduğundan, yürütmelerini diğer işlemlerle örtüştürmek (aynı anda yapmak) mümkündür. Böylelikle diğer yararlı işler aynı anda yapılarak bu işlemleri gerçekleştirme maliyeti gizlenebilmektedir.

Mart 2015'den önce CUDA akışlarının eşzamanlı olarak çalışma özelliği bulunmamaktaydı [29]. Birden fazla stream başlatılsa bile şekil 1.17.'deki gibi çalışmaktaydı. Ama bu tarihten sonra CUDA akışlar Şekil 1.18.'de olduğu gibi eşzamanlı çalışma özelliği kazanmıştır. Eğer donanım kaynakları yeterliyse, 3 farklı görev veya çekirdek (kernel) Şekil 1.18.'deki gibi aynı anda çalışabilecektir.

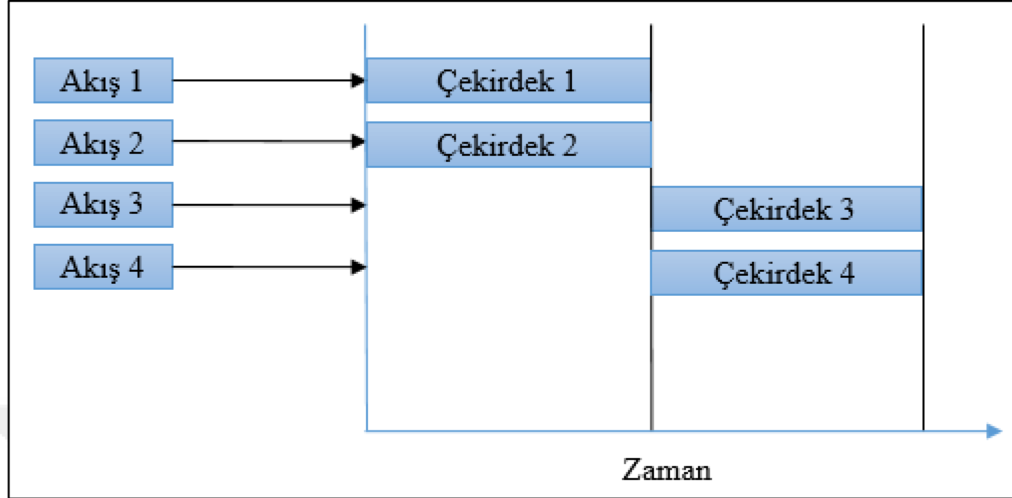


Şekil 1.18. Eşzamanlı akışlar

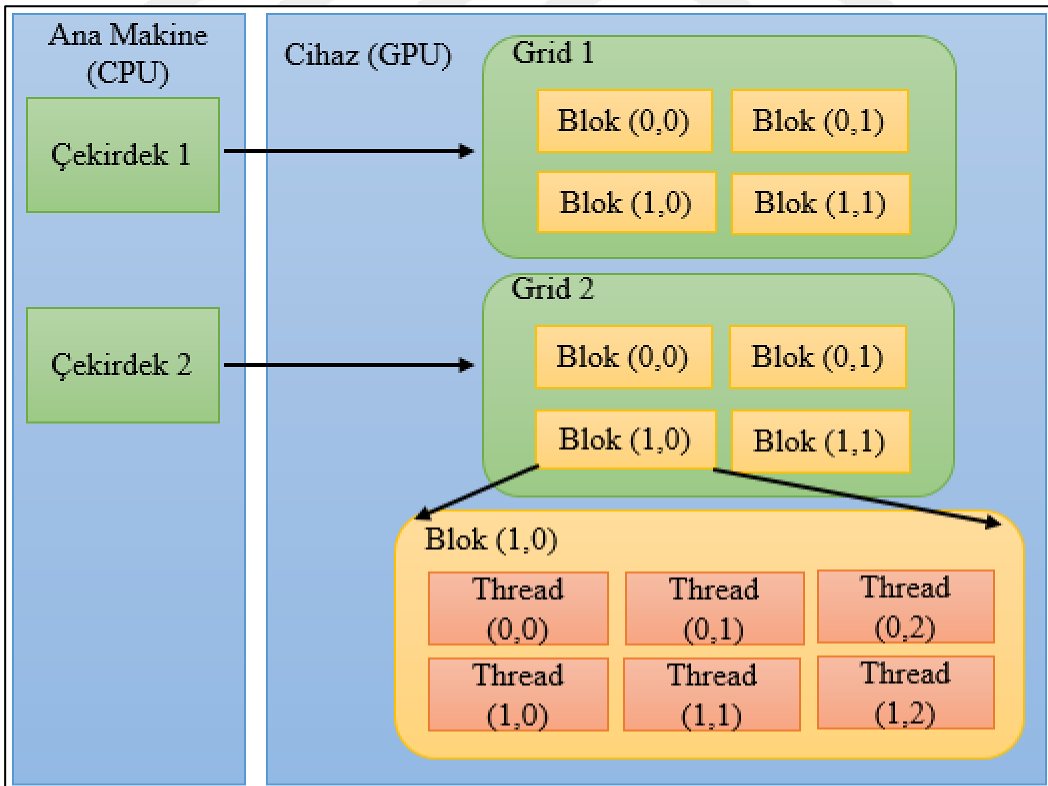
Bu özellikle birlikte aynı anda birden fazla problemin GPU üzerinde çözülebilmesi mümkün hale gelmiştir. Ya da bir problemin farklı parçalarını farklı stream'ler ile de çözmek mümkündür. Eşzamanlı çekirdeklerin maksimum sayısı, cihaza (GPU) bağımlıdır. Fermi mimarisine sahip cihazları 16 çekirdekli (kernel) eşzamanlılığı desteklerken ve Kepler mimarisine sahip cihazları 32 çekirdekli (kernel) eşzamanlılığı desteklemektedir. Eşzamanlı çekirdeklerin sayısı, paylaşılan bellek ve kaydediciler (register) gibi aygıtlardaki kullanılabilir kaynaklarla sınırlıdır. Eğer akışların kullandığı kaynaklar donanım limitleri aşarsa geri kalan akışlar Şekil 1.19.'daki gibi seri bir şekilde yürütülmektedir.

Thread açısından bakıldığında çoklu akış gösterimi Şekil 1.20.'deki gibidir. Şekil 1.20.'de görüldüğü gibi CUDA'nın eşzamanlı akış özelliği ile birden fazla çekirdek aynı

anda çalıştırılabilmektedir. Bu sayede aynı anda birden fazla problem GPU üzerinde yürütülebilmektedir. Ya da var olan bir problem birden fazla parçaya bölünebilmektedir.



Şekil 1.19. Kaynaklar yetmediğinde akışların davranışı



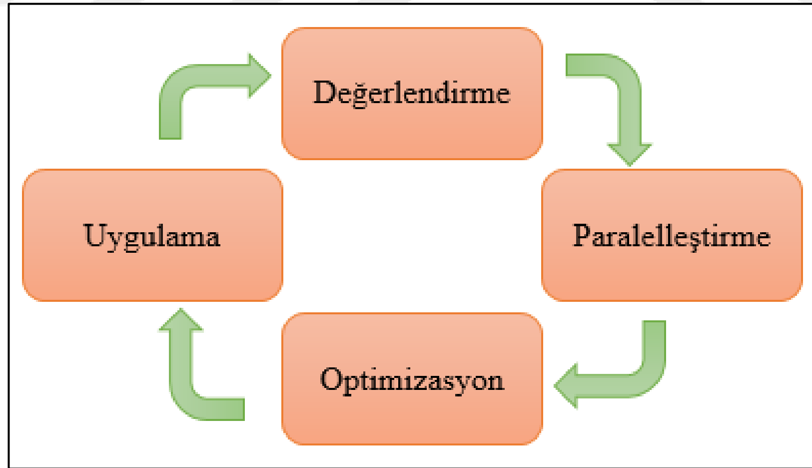
Şekil 1.20. Thread açısından çoklu stream yapısı

1.4.4. CUDA Uygulamasında Dikkat Edilmesi Gereken Noktalar

Günümüzdeki ihtiyaçlardan ötürü genel amaçlı hesaplamadan heterojen hesaplamaya doğru bir geçiş yaşanmaktadır. Paralel programlama daha önce hiç olmadığı kadar ilgi çekmektedir. Bundan dolayı paralel ve heterojen yazılımın verimli ve doğru bir şekilde nasıl uygulanacağını anlamak çok önemidir. Bu bölümde paralel programlama konusunda göz önünde bulundurulması gereken birkaç noktaya değinilecektir.

1.4.4.1. CUDA Geliştirme Süreci

Uygulamanın GPU'yu nasıl kullandığını anlamak, performans geliştirme fırsatlarını belirlemek için çok önemlidir. NVIDIA, geliştirme sürecini çekici ve keyifli hale getiren birçok güçlü ve kullanımı kolay araç sunmaktadır. DPOU (Değerlendirme, Paralelleştirme, Optimizasyon, Uygulama) [4], NVIDIA tarafından özel olarak CUDA'nın geliştirilmesi için sunulan yineleyici, özel bir geliştirme sürecidir. DPOU, Şekil 1.21'.deki dört aşamayla karakterize edilmektedir.



Şekil 1.21. CUDA uygulama geliştirme süreci

Değerlendirme aşaması, performans darboğazlarının veya yüksek hesaplama yoğunluğuna sahip kritik bölgelerin belirlenmesi için uygulamayı değerlendirmektir. Bu aşamada, CPU'yu tamamlamak için GPU'ların nasıl kullanılacağı değerlendirilmekte ve bu kritik bölgelerin hızlandırılması için stratejiler geliştirilmektedir. Değerlendirme safhasında,

kayda değer hesaplama içeren veri-paralel döngü yapılarına her zaman değerlendirme için daha yüksek öncelik verilmelidir. Bu tür bir döngü GPU ivmesi için ideal bir durumdur. Bu kritik bölgelerin tanımlanmasına yardımcı olmak ve uygulamadaki sorunlu noktaları ortaya çıkarmak için profil oluşturma araçları kullanmalıdır.

Paralelleştirme aşaması, bir uygulamanın darboğazı belirlendikten sonra kodun paralel hale getirilmesidir. Ana makine kodunu hızlandırmanın birkaç yolu vardır:

- CUDA paralel kütüphanelerini kullanma
- Paralelleştirici ve vektörize eden derleyicileri kullanma
- Paralelliği ortaya çıkarmak için manuel olarak CUDA çekirdekleri organize etmek

Bir uygulamayı paralel hale getirmek için en basit yaklaşım, mevcut GPU hızlandırılmış kitaplıklarını kullanmaktır. Uygulama BLAS veya FFTW gibi diğer C matematiksel kitaplıklarını zaten kullanıyorsa cuBLAS veya cuFFT gibi CUDA kitaplıklarını kullanmak için kolayca geçiş yapılabilmektedir.

Ana bilgisayar kodlarını paralelleştirmeye yönelik nispeten zahmetsiz diğer bir yaklaşım, paralelleştirici derleyicileri kullanmaktır. OpenACC açıkça hızlandırıcı ortamlar için tasarlanmış açık kaynak, standart derleyici yönergeleri kullanmaktadır. OpenACC uzantıları, verilerin işlenen öğelerin yakınında bulunmasını ve bir dizi derleyici direktifi sağlanmasını sağlamak için yeterli kontrol sağlamaktadır. Bu yapılar paralel ve çok çekirdekli işlemcilerde GPU programlamasını basit ve taşınabilir yapmaktadır.

Uygulama tarafından istenen işlevsellik veya performansın var olan paralel kütüphanelerden veya paralelleştirici derleyicilerin sağlayabileceğinden fazla olduğu durumlarda, paralelleştirme için CUDA C ile çekirdeklerin yazılması zorunlu hale gelmektedir. CUDA C, GPU'ların paralel gücünden tam olarak faydalanma yeteneğini en üst düzeye çıkarmaktadır.

Optimizasyon aşaması, kodu paralel çalışacak şekilde düzenledikten sonra performansı artırmak için uygulamanın optimize edilmesidir. CUDA tabanlı optimizasyonlar genelde aşağıdaki iki seviyede uygulanabilmektedir:

- Grid düzeyi
- Çekirdek düzeyi

Grid düzeyi optimizasyonu sırasında, odaklanma genel GPU kullanımı ve verimlilik üzerinedir. Grid düzeyinde performansı en iyi duruma getirme teknikleri arasında birden çok çekirdeği aynı anda çalıştırmak ve CUDA akışları ve etkinlikleri kullanarak çekirdek

yürütmeyi ve veri aktarmalarını üst üste çakıştırmak sayılabilir. Bir çekirdeğin performansını sınırlayabilecek üç temel faktör vardır:

- Bellek bant genişliği
- Hesaplama kaynakları
- Komut ve bellek gecikmesi

Çekirdek düzeyi optimizasyon sırasında, GPU bellek bant genişliğinin verimli kullanılması ve hesaplama kaynakları ile komut ve bellek gecikmelerinin azaltılması veya gizlenmesi üzerinde odaklanılmaktadır.

Uygulama aşaması, GPU hızlandırmalı uygulamanın doğru sonuçlar verdiğini doğruladıktan sonra, GPU bileşenlerini kullanarak sistemin nasıl uygulanılacağını düşünmektir. Örneğin, bir CUDA uygulaması uygulanırken, hedef makine CUDA özellikli bir GPU'ya sahip olmamasına rağmen düzgün çalışmaya devam etmesini sağlamak gerekebilir. CUDA çalışma zamanı (runtime) api, CUDA özellikli GPU'ları algılar ve donanım ve yazılım yapılandırmasını kontrol etmeyi sağlayan çeşitli işlevler sunmaktadır. Bununla birlikte, uygulama algılanan donanım kaynaklarına manuel olarak uyumlu olmalıdır.

1.4.4.2. Profil Destekli Optimizasyon

CUDA, grid ve çekirdek düzeyinde performans fırsatlarını belirlemeye yardımcı olacak aşağıdaki yararlı ve güçlü araçları sunmaktadır:

- Nsight Eclipse (nsight) [30]
- NVIDIA Görsel Profil (Visual Profiler - nvvp) [31]
- NVIDIA Komut Satırı Profil (Command-line Profiler - nvprof) [32]

Bu profil oluşturma araçları, optimizasyon işlemleri sırasında geliştiriciyi yönlendirirken etkili olmakta ve performansı artırmak için en iyi eylem yolunu önermektedir. Profil odaklı optimizasyon, profil bilgilerine dayanarak programınızı optimize etmek için yinelemeli bir işlemdir. Tipik olarak, aşağıdaki yinelemeli yaklaşımı kullanılır:

1. Bilgi toplamak için görsel profil bir uygulamaya uygulanır.
2. Uygulamadaki sorunlu yerler belirlenir.
3. Performans engelleyicileri tespit edilir.
4. Kod optimize edilir.
5. İstenilen performans sağlanıncaya dek önceki adımlar tekrarlanır.

En önemli adım, performansı engelleyen noktaları tanımlamaktır. CUDA profilleme araçları, koddaki inhibitörleri bulmaya yardımcı olmaktadır. Bir çekirdek için en olası performans engelleyiciler şunlardır:

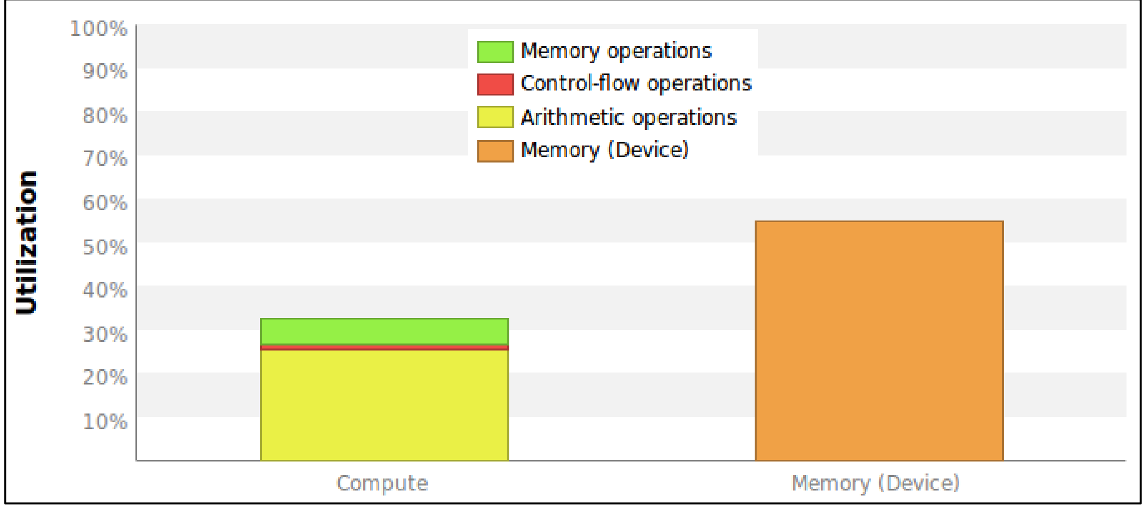
- Bellek bant genişliği
- Komut işleme hızı
- Gecikme (aritmetik ve bellek)

NVIDIA Görsel Profil (Visual Profiler - nvvp) ile Şekil 1.22.'deki gibi programın bağımlılık analizi yapılarak, hangi noktaların ne kadar zaman harcadığı görebilmektedir.

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
calculate_forces(float4*, float4*)	74.98 %	338.405 ms	0 ns
cudaMalloc	13.24 %	59.754 ms	0 ns
cudaDeviceReset	6.49 %	29.285 ms	0 ns
<Other>	5.14 %	23.193 ms	0 ns
cuDeviceGetAttribute	0.05 %	227.372 µs	0 ns
cudaGetDeviceProperties	0.05 %	213.381 µs	0 ns
cudaFree	0.02 %	98.555 µs	0 ns
cuDeviceTotalMem_v2	0.02 %	91.989 µs	0 ns
[CUDA memcpy HtoD]	0.01 %	36.811 µs	0 ns
cuDeviceGetName	0.01 %	25.872 µs	0 ns
[CUDA memcpy DtoH]	0.00 %	17.286 µs	0 ns
cudaSetDevice	0.00 %	4.641 µs	0 ns
cuDeviceGetCount	0.00 %	1.542 µs	0 ns
cudaGetLastError	0.00 %	1.388 µs	0 ns
cuDeviceGet	0.00 %	713 ns	0 ns
cudaMemcpy	0.00 %	0 ns	17.286 µs
cudaConfigureCall	0.00 %	0 ns	0 ns
cudaSetupArgument	0.00 %	0 ns	0 ns
cudaLaunch	0.00 %	0 ns	0 ns
cudaDeviceSynchronize	0.00 %	0 ns	338.405 ms

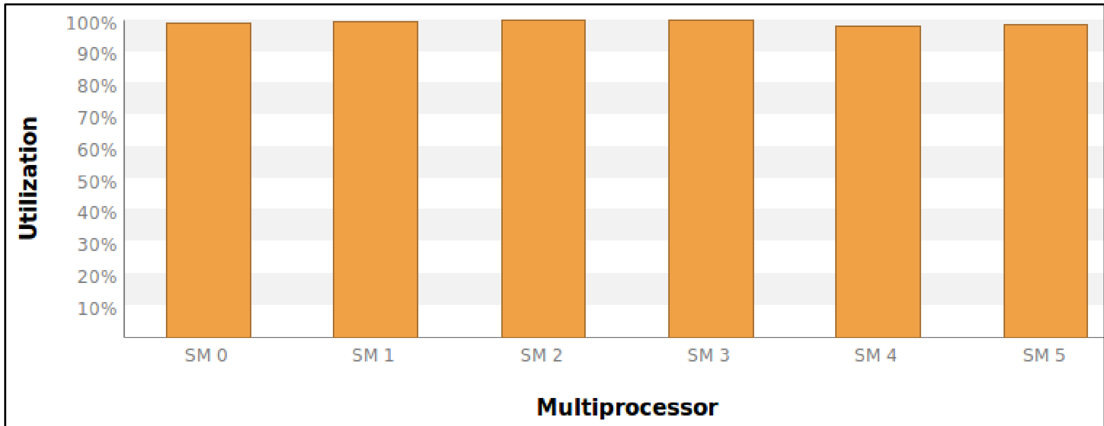
Şekil 1.22. CUDA programının bağımlılık analizi

NVIDIA Görsel Profil ile Şekil 1.23.'deki gibi hesaplama gücünden ve bellekten ne kadar yararlanıldığı görebilmektedir. Ayrıca performansı neyin sınırladığı yine buradan incelenebilmektedir.



Şekil 1.23. CUDA çekirdek (kernel) performansını kısıtlayıcı faktörler

Thread blokları GPU işlemcilerde (multiprocessor) çalıştırılmak üzere dağıtılmaktadır. Blok sayısına ve çalışma sürelerine bağlı olarak çekirdeğin çalışması boyunca bazı işlemciler (multiprocessor) diğerlerine göre daha fazla kullanılabilmektedir. NVIDIA Görsel Profil ile Şekil 1.24.'deki gibi hangi işlemciden (multiprocessor) ne kadar yararlandığı tespit edilebilmektedir.



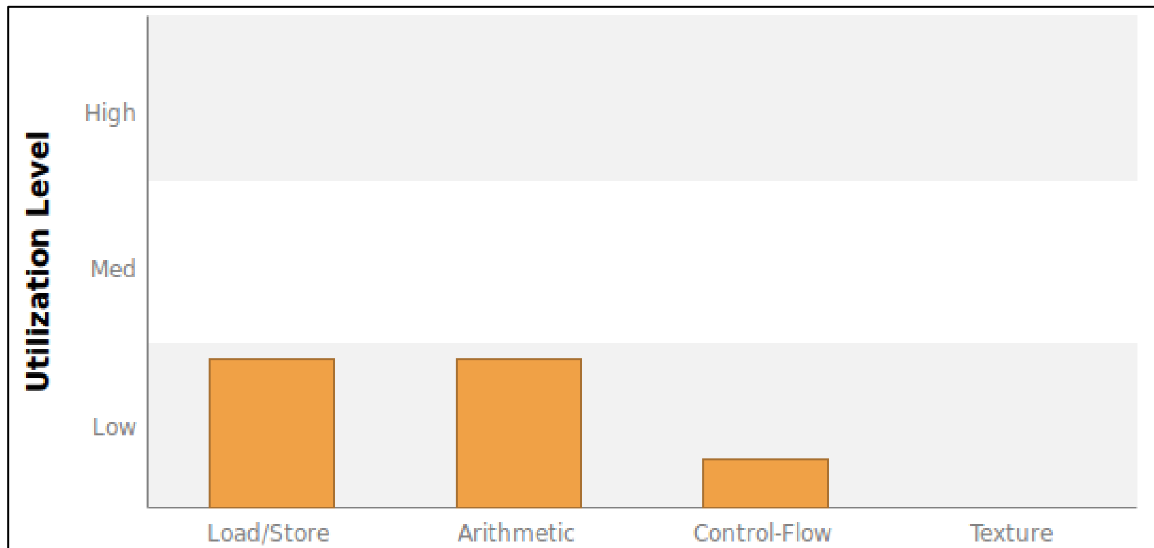
Şekil 1.24. GPU'da akışlı çoklu işlemci (streaming multiprocessor - SM) kullanımı

NVIDIA Görsel Profil ile Şekil 1.25.'deki gibi thread'lerin, warp'ların, kaydedicilerin (register) ve paylaşımlı belleğinin ne kadarından faydalanıldığı görülebilmektedir.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [12,1,1] (12 blocks)Block Size: [1(
Occupancy Per SM				
Active Blocks		2	16	0 2 4 6 8 10 12 14 16
Active Warps	61.19	64	64	0 10 20 30 40 50 6064
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	95.6%	100%	100%	0% 50% 100%
Warps				
Threads/Block		1024	1024	0 256 512 768 1024
Warps/Block		32	32	0 5 10 15 20 25 3032
Block Limit		2	16	0 2 4 6 8 10 12 14 16
Registers				
Registers/Thread		25	65536	0 16384 32768 49152 65536
Registers/Block		32768	65536	0 32k 64k
Block Limit		2	16	0 2 4 6 8 10 12 14 16
Shared Memory				
Shared Memory/Block		0	49152	0 16k 32k 48k
Block Limit			16	

Şekil 1.25. Thread, warp, kaydedici ve paylaşımlı bellek kullanımı

NVIDIA Görsel Profil ile Şekil 1.26.’daki gibi çekirdeğin çalışması boyunca GPU birimlerinden ne ölçüde faydalanıldığı incelenebilmektedir.

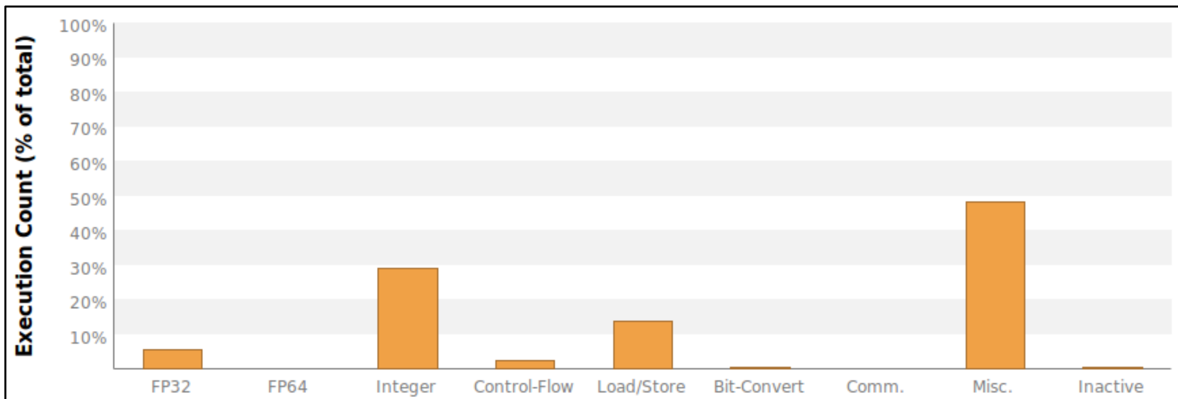


Şekil 1.26. Çekirdeğin çalışması boyunca GPU birimlerinden faydalanma oranı

NVIDIA Görsel Profil ile Şekil 1.27.’deki gibi çeşitli bellek türlerinden ne ölçüde faydalanıldığı detaylı bir şekilde incelenebilmektedir. Yine Şekil 1.28.’deki gibi çalıştırılan komutların toplam komutlara oranı incelenebilmektedir.

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	242755126	74.733 GB/s	
Local Stores	218443604	24.559 GB/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	28318848	2.55 GB/s	
Global Stores	6144	2.212 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	489523722	101.845 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	679028116	61.105 GB/s	
L1 Writes	546581860	49.186 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	1225609976	110.291 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	331931328	29.87 GB/s	
Writes	546363588	49.166 GB/s	
Total	878294916	79.036 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	17	1.529 kB/s	Idle Low Medium High Max
Writes	1	89 B/s	Idle Low Medium High Max

Şekil 1.27. Bellek türlerinin kullanımı



Şekil 1.28. Çalıştırılan komutların toplam komut sayısına oranları

1.5. N-Cisim Simülasyonu

N-Cisim simülasyonu, cisimlerden oluşan dinamik bir sisteminin simülasyonudur. Bu simülasyonda her cisim diğer cisimler ile sürekli etkileşim halindedir. N-Cisim simülasyonları, gezegen sistemleri, küresel kümeler, galaksiler, galaksilerin kümeleri ve evrendeki diğer büyük ölçekli yapılar gibi çeşitli astronomik sistemlerin oluşumunu ve evrimini araştırmak için yaygın şekilde kullanılmaktadır [33, 34].

N-Cisim simülasyonunun kullanıldığı tek alan astronomi değildir. Bilgisayar grafikleri, moleküler dinamik [35, 36], akışkanlar mekaniği [37, 38], elektromanyetik [39] gibi hesaplamalı bilimlerin birçok yerinde kullanılmaktadır. N-Cisim simülasyonu hesaplamalı bilimlerdeki en zor problemlerden biridir. N-Cisim simülasyonu üzerine yapılan çalışmalar ACM (Association for Computing Machinery) tarafından verilen Gordon Bell ödülünü birçok kez kazanmıştır [40, 41, 42, 43, 44, 45].

Astronomik N-Cisim simülasyonlarında, gezegenler, yıldızlar ya da galaksiler birbirleriyle etkileşen cisimler olarak kabul edilmektedir. Cisimler arasındaki etkileşimler sayısal olarak Eşitlik 1.1'deki Newton'un evrensel çekim yasasına göre değerlendirilmektedir.

$$F_i \approx Gm_i \, . \sum_{1 \le j \le N} \frac{m_j r_{ij}}{\left(\left\| r_{ij} \right\|^2 + \varepsilon^2 \right)^{3/2}} \tag{1.1}$$

Eşitlik 1.1'de m_i ve m_j i ve j cisminin kütlelerini, $r_{ij} = x_j - x_i$ cisim i'den cisim j'ye olan uzaklık vektörünü, ε yumuşatma katsayısını (softening factor) [46, 47] ve G yerçekimi sabitini göstermektedir.

Astrofizik simülasyonlar çarpışmalı ve çarpışmasız olarak ikiye ayrılmaktadır. Çarpışmasız simülasyonlarda cisimler arasındaki çarpışmalar engellenmektedir. Eğer cisimler birbirinin içinden geçen galaksileri temsil ediyorsa bu engelleme durumu gayet normaldir. Bu koşulu sağlamak için, yumuşatma faktörü $\varepsilon^2 > 0$ paydaya eklenmektedir. Böylelikle uzaklık vektörünün sıfırdan büyük olması yani iki cisim arasında daima belli bir uzaklık olması garanti edilerek çarpışmasız bir simülasyon elde edilmektedir. Cisimler Eşitlik 1.2'deki Newton'un hareket yasasına ve Eşitlik 1.3 ve Eşitlik 1.4'deki İleri Euler (Forward Euler) metoduna [48] göre hareket ettirilmektedir.

$$a_i = F_i / m_i \quad (1.2)$$

$$v_{k+1} = v_k + a_k dt \quad (1.3)$$

$$r_{k+1} = r_k + v_k dt \quad (1.4)$$

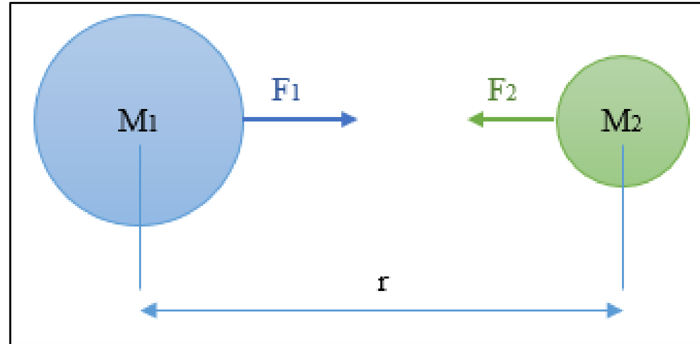
Eşitlik 1.2'de m_i i cisminin kütlesini, F_i i cismine etki eden net kuvveti ve a_i i cismin ivmesini göstermektedir. Eşitlik 1.3'de ve Eşitlik 1.4'de v_k cismin mevcut hızını r_k ise cismin mevcut konumu göstermektedir. $dt = t_{k+1} - t_k$ simülasyondaki iki adım arasındaki zaman farkına eşittir. İleri Euler birinci derece integral yöntemidir. Hata oranı yüksek olsa da uygulama açısından basit bir yöntemdir. Bu yüzden uygulamamızda bu yöntem tercih edilmiştir.

Uygulamalardaki integral yöntemi seçimi çalışılan sistemin doğasına bağlı olmaktadır [49]. Yüksek dereceli integral yöntemleri daha fazla zaman alsa da karmaşıklığı O (N)'dir ve N cisim sayısı büyüdükçe hesaplama zamanı maliyeti önemsiz olmaktadır.

1.5.1. N-Cisim Simülasyonundaki Karşılıklı Kuvvetler

N-Cisim simülasyonunda Eşitlik 1.5'de ve Şekil 1.29.'da gösterildiği gibi iki cisim arasındaki kütle çekim kuvveti büyüklük açısından birbirine eşit ve yön olarak birbirinin tersidir. N-Cisim simülasyonunda bu eşitliğe karşılıklı kuvvetler (reciprocal forces) denir. Bu optimizasyon kullanılarak kaba kuvvet yöntemindeki kuvvet hesaplama işlemi iki kat azaltılabilir.

$$F_{12} = -F_{21} = G\frac{m_1 \times m_2}{r^2} \quad (1.5)$$



Şekil 1.29. İki cisim arasındaki çekim kuvveti

1.5.2. Literatür Araştırması

Literatürde, N-Cisim simülasyonları ile ilgili çalışmalar, yazılım ve donanım olmak üzere iki gruba ayrılmaktadır. Yazılım bölümü kaba kuvvet yöntemleri (brute force) ve hiyerarşik yöntemler olarak ikiye ayrılmaktadır. Bütün-çiftler (All-pairs) metodu gibi kaba kuvvet yöntemlerin hesaplama karmaşıklığı $O(N^2)$ olduğundan problemin çözümü çok fazla gerektirmektedir. Bundan dolayı, bilimsel araştırmanın büyük çoğunluğu hiyerarşik yöntemlere odaklanmaktadır [50]. Hiyerarşik yöntemler, kaba kuvvet yöntemlerinden farklı olarak, cisimler arasındaki tüm etkileşimleri değerlendirmemektedir. Genel olarak hiyerarşik yöntemler alanı belirli ölçütlere göre hiyerarşik seviyelere bölmekte ve bu seviyeler arasındaki etkileşimleri değerlendirmektedir. Literatürdeki önemli hiyerarşik yöntemler ve bu yöntemlerin karmaşıklıkları Tablo 1'de verilmektedir.

Tablo 1.1 Hiyerarşik yöntemler ve karmaşıklıkları

Hiyerarşik Yöntemler	**Karmaşıklıkları**
Ağaç kodu [51]	O(N log N)
Hızlı çok kutuplu yöntemler (FMM) [52]	O(N log N) – O(N)
Parçacık-Izgara (PM) kodu [53]	O(N log N)
Paçacık^3-Izgara(P^3M) [53]	O(N log N)
Parçacık-Izgara-Ağaç kodu [54]	O(N log N)

Donanım kısmı ise özel olarak tasarlanmış CPU'lar ve GPU kullanımı olarak ikiye ayrılmaktadır. N-Cisim problemi için tasarlanan ilk CPU 1990 üretilen GRAPE-1'dir [55] ve performansı 100 Mflop/s'dir. N-Cisim problemini çözmek için bu CPU'lardan bir küme (cluster) oluşturulmaktadır. 2012'de üretilen en yeni GRAPE-8 [56], 48 veri yoluna ve 480 Gflop/s işleme gücüne sahiptir. GPU programlamanın geçmişi 1990'nın sonlarına uzanmaktadır. Fakat yazılımı ve donanımı birleştirerek GPU'ların genel amaçlı hesaplama için kullanılabilmesini ilk defa NVIDIA, 2006 yılında CUDA'yı ortaya çıkararak başarmıştır. Bu yeni programlama paradigmasından yararlanabilecek az sayıdaki bilimsel problemlerden biri de N-Cisim problemidir. N-Cisim probleminin CUDA'ya nasıl uygulanabileceğini ve hangi parametrelerin performansı artırabileceğini inceleyen ilk

çalışma 2007'de yapılmıştır [49]. Bu çalışmada herhangi bir algoritmik iyileştirme söz konusu değildir. Bütün-çiftler yönteminin uygulaması şeklinde gerçekleştirilmiştir.

N-Cisim problemin ile ilgili yapılan çalışmaların büyük çoğunluğu hiyerarşik yöntemler üzerine olduğundan CUDA ile ilgili yapılan çalışmaların büyük bir çoğunluğu da yine bu hiyerarşik yöntemlerin CUDA ile GPU üzerinde uygulanması şeklinde olmuştur [57, 58, 59, 60, 61, 62].

CUDA'nın piyasaya sürülmesiyle birlikte, bu tür bilimsel problemlerde GPU'ların hızlı kullanımını etkileyen iki önemli faktör olmuştur. Birincisi, GPU'lar GRAPE donanımına benzer bir yapıda olması, bu nedenle var olan kodda herhangi bir önemli değişiklik olmadan GPU'da uygulanabilmesiydi [63]. İkincisi GRAPE'in aksine GPU'lar kitlesel bir üretim ürünü olduğu için herkes tarafından erişilebilir olmasıydı. Yapay zeka çalışmalarında kullanılabilme kabiliyeti sayesinde CUDA'nın popülaritesi artmaya devam etmektedir.

Donanım kısmındaki teknikler genel olarak veri büyüklüğünün küçük olduğu problemlerde kullanılmaktadır. Veri boyutunun çok büyük olduğu durumlarda ise süper bilgisayarlar kullanılmaktadır. Örneğin 2010 yılında yapılan The Millennium-XXL projesinde [64] veri büyüklüğü 300 milyardan fazladır. Zamanında en güçlü 15 süper bilgisayar içinde olan Juropa'da yürütülmüştür. Simülasyonun gerçekleştirilebilmesi için 300 yıllık CPU zamanına denk performans gerekmiştir. 12 binden fazla bilgisayar çekirdeği ve 30 terabayt bellek kullanılmıştır. Simülasyon sonuncunda 100 terabaytlık veri üretilmiştir. Simülasyonun bitmesi yaklaşık 45 gün sürmüştür. The Millennium-XXL projesinde hiyerarşik ağaç kodunu temel alan hibrit bir yöntem kullanılmıştır [65].

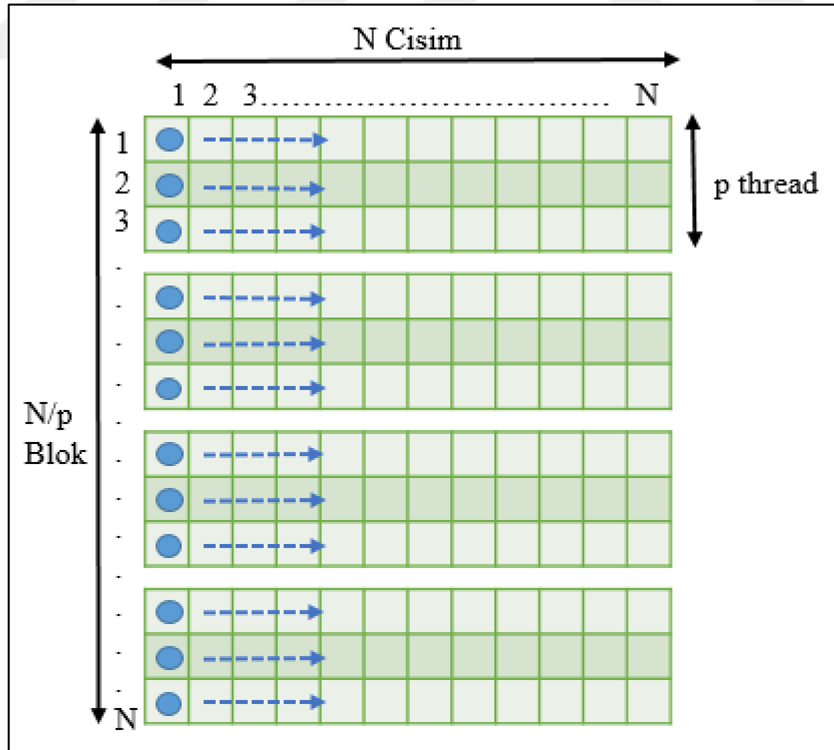
2. YAPILAN ÇALIŞMALAR

2.1. Giriş

Bu bölümde, Bütün-Çiftler Algoritması (BÇA), Karşılık Kuvvetler Optimizasyonu (KKO) ve Blokları Düzenlenmiş Karşılıklı Kuvvetler Optimizasyonu (BDKKO) CUDA ile uygulanmıştır. Karşılıklı kuvvetler ve Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonu CUDA Streams özelliği kullanarak ayrıca uygulanmıştır. Son olarak bu algoritmalar başlıca optimizasyon teknikleri uygulanarak optimize edilmeye çalışılmıştır.

2.2. Bütün-Çiftler(All-Pairs) Yönteminin CUDA ile Gerçekleştirilmesi

Bütün-çiftler yöntemi şekil 2.1.'de gösterildiği gibi uzunluğu N olan bir veri için NxN'lik bir yapıdaki her f_{ij} çiftinin hesaplanmasıdır.



Şekil 2.1. Bütün-çiftler yönteminin CUDA ile gerçekleştirilmesi

Hesaplamalar yapılırken, cisim i'ye etki eden toplam net kuvveti bulmak için i'ninci satırdaki tüm kuvvetler toplanmaktadır. Daha sonra bölüm birde bahsedilen Eşitlik 1.1 ile cismin ivmesi, Eşitlik 1.2 ile cismin hızındaki değişim ve son olarak da Eşitlik 1.3 ile bir sonraki adımda cismin nerede olacağı hesaplanmaktadır.

Büyük problemlerde p thread sayısı N cisim sayısından çok küçüktür. Şekil 2.1.'de görüldüğü thread'ler soldan sağa paralel olarak çalışsa da yukarıdan aşağıya doğru bloklar halinde sıralı olarak çalışmaktadırlar.

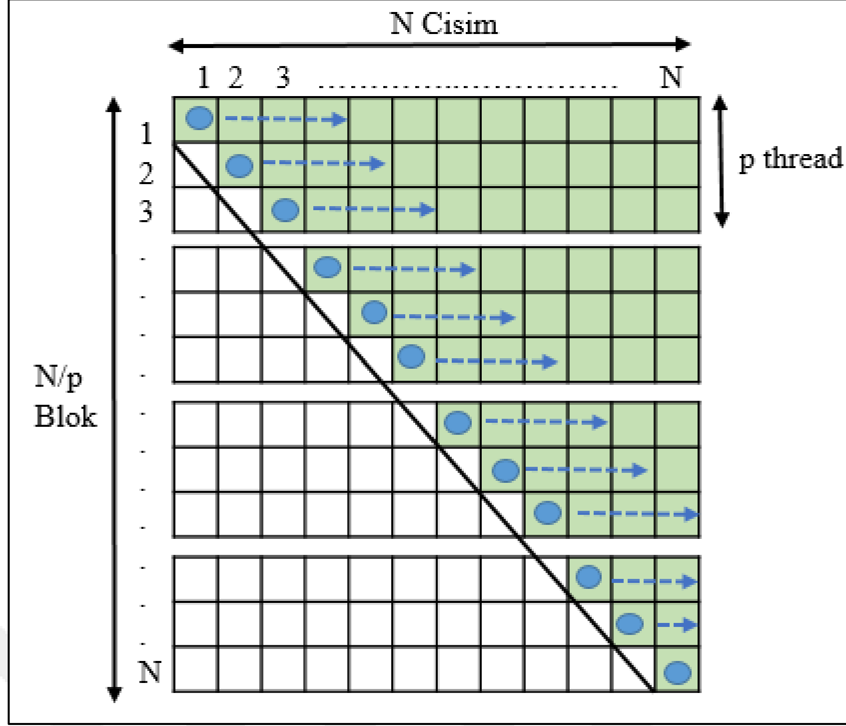
Yukarıda bahsedildiği gibi toplam net kuvveti bulmak için hesaplama yapılırken hesaplanan kuvvetlerin aynı anda toplanması gerekmektedir. Çünkü Şekil 2.1.'de olduğu gibi NxN'lik bir sonuç dizisi tutmak olası değildir. Örnek olarak N veri büyüklüğü bir milyon olduğunda böyle bir sonuç dizisini tutmanın maliyeti 12 Terabayt olmaktadır. Böyle bir kaynağa standart bir bilgisayarda sahip olmak mümkün değildir.

2.3. Karşılıklı Kuvvetler Optimizasyonunun CUDA ile Gerçekleştirilmesi

Bölüm birde Eşitlik 1.5'i karşılıklı kuvvetler olarak adlandırmıştık. Şekil 2.2.'de görüldüğü gibi bu optimizasyonu kullanarak kuvvet hesaplamasını 2 kat azaltılabilmektedir.

Bu optimizasyonu CUDA ile uygulamanın zorluğu Şekil 2.2.'de olduğu gibi her sütunda aynı anda bir thread'in çalışması gerekmektedir. Eğer bu koşul sağlanmaz ise aynı global bellek adresine çoklu yazma hatası oluşmaktadır. Örneğin birinci thread F_{15}'i hesaplarken üçüncü thread F_{35}'i hesaplarsa bu iki thread'in karşılıklı kuvvetleri F_{51} ve F_{53} olur. Hesaplamalar tamamlandıktan sonra F_{51} ve F_{53} beşinci global bellek adresinde depolanacaktır. Sonuç olarak aynı global adrese çoklu yazma hatası oluşacaktır.

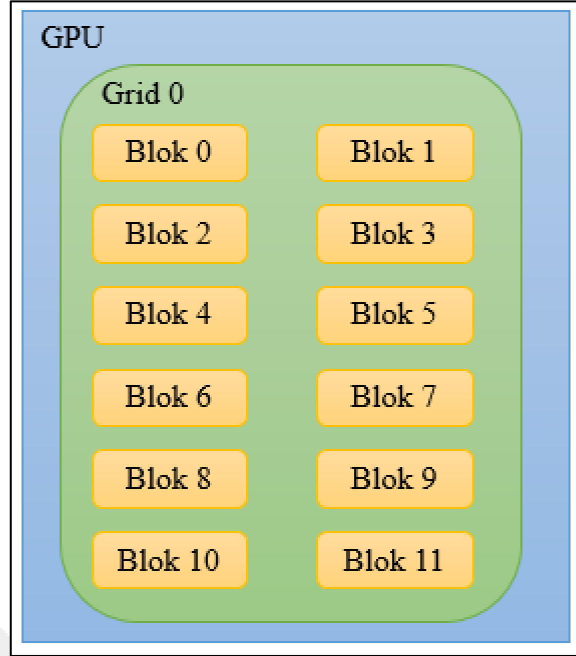
Aynı adrese çoklu yazma hatasının üstesinden gelmek için thread'leri senkronize edebilmemiz gerekmektedir. CUDA bir bloğun içindeki thread'leri "__syncthreads()" fonksiyonu ile senkronize edebilir fakat bloklar arasındaki thread'leri organize etme özelliği bulunmamaktadır. Bu sorun, kullanılan her blok için bir sonuç dizisi tanımlanarak çözülebilmektedir. Programımız Şekil 2.3'de görüldüğü gibi 1 grid ve 12 bloktan oluştuğu için 12 adet sonuç dizisi tanımlayarak aynı adrese çoklu yazma hatasının oluşumunun önüne geçilebilmektedir.



Şekil 2.2. Karşılıklı kuvvetler optimizasyonunun CUDA ile uygulaması

2.4. KKO'nun CUDA Stream Özelliği Kullanılarak Gerçekleştirilmesi

Aynı adrese çoklu yazma sorununu CUDA 7 ile birlikte gelen Stream özelliğini kullanarak da çözülebilmektedir. Stream özelliği kullanılarak programın yapısı değiştirilip bloklar arasındaki thread senkronizasyonun üstesinden gelinebilmektedir. Şekil 2.3'deki gibi bir akış içinde birden fazla blok oluşturmaktansa, Şekil 2.4'deki gibi her akış içinde bir blok olmak üzere birden çok akış oluşturabilir. Her akış içinde bir blok ve her akışın kendi global bellek adresi olduğundan aynı global bellek adresine çoklu yazma hatası, problemin yapısı gereği ortadan kalkmaktadır. Ve böylelikle algoritma açısından herhangi bir karmaşıklık oluşturmadan aynı adrese çoklu yazma sorunu çözülebilmektedir.



Şekil 2.3. Streams kullanılmadığında grid ve blok yapısı

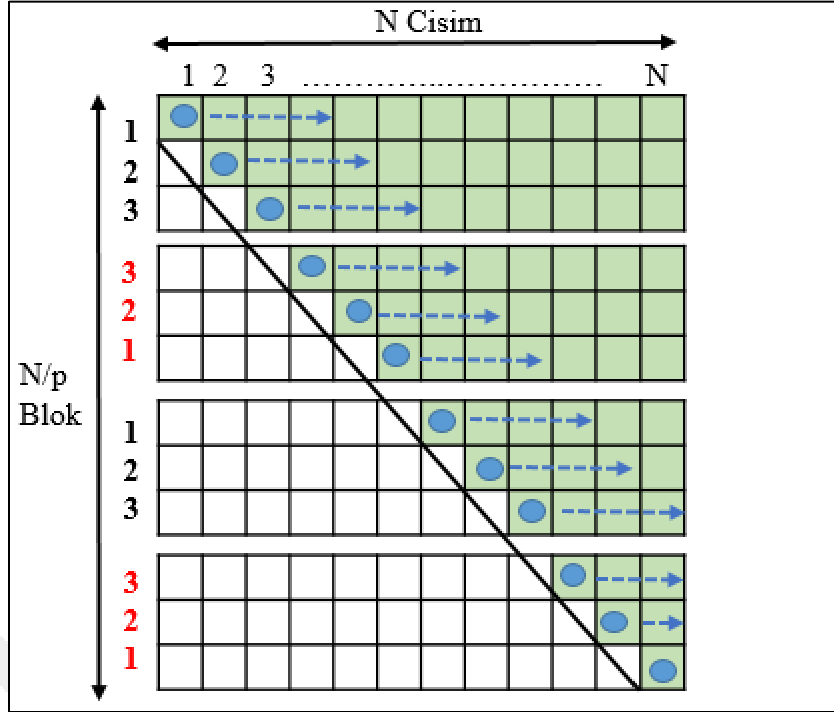
Şekil 2.4. Streams kullanıldığında grid ve blok yapısı

2.5. KKO’da Bloklar Arası İş Yükünün Düzenlenmesi

Karşılıklı kuvvetler optimizasyonunu NVIDIA Görsel Profil aracılığıyla analiz edildiğinde Şekil 2.5’deki gibi bir sonuç ortaya çıkmaktadır. Karşılıklı kuvvetler optimizasyonunda bloklar Şekil 2.2’deki gibi eğimli bir yapıda ilerlediğinden birinci blok hep ilk sıraya gelmektedir. Bunun sonucunda birinci blok her defasında en uzun veriyle işlem yapmak zorunda kalmaktadır. Bundan dolayı diğer bloklar işlerini erken bitirmekte ve birinci bloğun işini bitirmesini beklemektedirler. Dolayısıyla programın başarısı birinci bloğun başarısına bağlı kalmaktadır.

Bloklar arasında dengeli iş yükü dağıtımı yapmak için her adımda bloklar tersine çevrilebilir. Şekil 2.6’daki gibi birinci blok ilk adımda birinci sıradayken ikinci adımda son sırada olmaktadır. Böylelikle bloklar arası iş yükü dağıtımı Şekil 2.7’deki gibi eşit olarak yapılabilmektedir. Bu bloklar arası düzenleme hem Stream özelliği kullanılarak hem de Stream özelliği kullanılmadan gerçekleştirilmiştir.

Şekil 2.5. Karşılıklı kuvvetler optimizasyonunda blokların çalışma zamanları



Şekil 2.6. Karşılıklı kuvvetler optimizasyonunda blokların düzenlenmesi

Blok 1	calculate_forces(float4*, float4*, int)
Blok 2	calculate_forces(float4*, float4*, int)
Blok 3	calculate_forces(float4*, float4*, int)
Blok 4	calculate_forces(float4*, float4*, int)
Blok 5	calculate_forces(float4*, float4*, int)
Blok 6	calculate_forces(float4*, float4*, int)
Blok 7	calculate_forces(float4*, float4*, int)
Blok 8	calculate_forces(float4*, float4*, int)
Blok 9	calculate_forces(float4*, float4*, int)
Blok 10	calculate_forces(float4*, float4*, int)
Blok 11	calculate_forces(float4*, float4*, int)
Blok 12	calculate_forces(float4*, float4*, int)

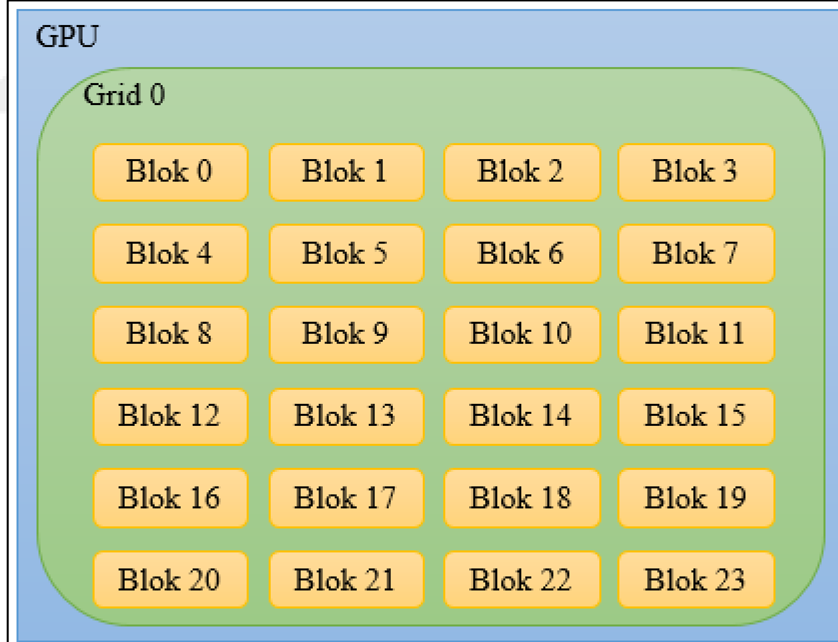
Şekil 2.7. Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonun çalışma zamanı

2.6. Algoritmaların Temel Optimizasyon Yöntemleri ile Optimize Edilmesi

Bu bölümde paralel programlamadaki başlıca optimizasyon tekniklerinden olan Farklı yapıdaki grid ve blok yapısı, Paylaşımlı bellek kullanımı ve Döngü açma (Loop unrolling) yöntemi Bütün-çiftler, Karşılıklı kuvvetler ve Blokları düzenlemiş karşılıklı kuvvetler algoritmaları üzerinde uygulanmıştır.

2.6.1. Farklı Yapıdaki Grid ve Blok Yapısı

Bütün-çiftler, Karşılıklı kuvvetler ve Blokları düzenlemiş karşılıklı kuvvetler algoritmalarının thread organizasyonu ilk olarak Şekil 2.3'deki 12 blok ve her blokta 1024 thread olacak şekilde düzenlenmiştir. Daha sonra thread organizasyonu Şekil 2.8'deki gibi 24 blok ve her blokta 512 thread olacak şekilde düzenlenmiştir. Bu değişikliğin çalışma zamanına (runtime) nasıl etki ettiği incelenmiştir.



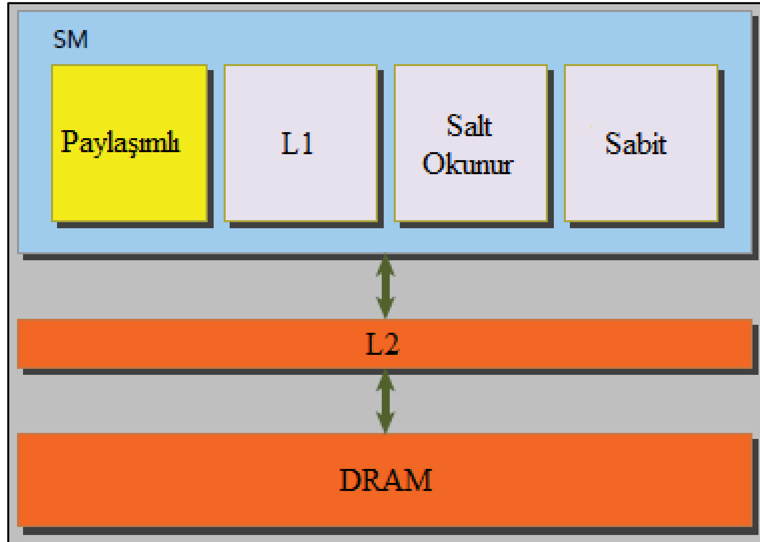
Şekil 2.8. 24 x 512 thread organizasyonu

2.6.2. Paylaşımlı Bellek Kullanımı

Paylaşımlı bellek GPU'nun anahtar parçalarından biridir. Fiziksel olarak, her SM o anda SM üzerinde çalıştırılan thread bloğundaki tüm thread'ler tarafından paylaşılan küçük boyutlu ve düşük gecikmeli bir bellek havuzu içermektedir. Paylaşımlı bellek, aynı thread bloğunun içindeki thread'lerin iş birliği yapabilmesine, yonga (on-chip) üzerindeki verinin tekrar kullanımının kolaylaştırılmasına ve global bellek bant genişliğinin önemli ölçüde azaltılmasına olanak sağlamaktadır. Paylaşımlı bellek açıkça uygulama tarafından kontrol edilebildiğinden program tarafından yönetilen önbellek olarak tanımlanmaktadır.

Şekil 2.9.'da görüldüğü gibi bütün yükleme ve depolama işlemleri L2 önbelleği üzerinden geçmektedir. Bu yüzden SM arasındaki veri birleşiminin en önemli noktası L2 önbelleğidir. Şekil 2.9.'da görüldüğü gibi paylaşımlı bellek ve L1 önbelleği , L2 önbelleği ve global belleğe göre SM'lere fiziksel olarak daha yakındır. Bu yakınlık, yaklaşık olarak 20-30 kat daha az bellek gecikmesi ve 10 kat daha fazla bant genişliği anlamına gelmektedir.

Paylaşımlı bellek kullanımı sadece Bütün-çiftler algoritmasına uygulanmıştır. Karşılıklı kuvvetler ve Blokları düzenlenmiş karşılıklı kuvvetler algoritmaları Şekil 2.2.'deki yapıları gereği paylaşımlı bellek kullanımına elverişli değildir.



Şekil 2.9. GPU'da paylaşımlı bellek gösterimi

2.6.3. Döngü Açma (Loop Unrolling) Tekniği

Döngü açma tekniği temel bir derleyici optimizasyonudur. Amaç döngü ek yükünü (overhead) azaltmaktır. CPU'lar sıralı ifadeleri kontrol işlemlerine göre daha hızlı şekilde yürütmektedirler. Şekil 2.10.'daki gibi döngü özyineleme (iteration) sayısı azaltılarak ve döngü gövdesindeki kod tekrarlanarak, kontrol sayısı azaltılmakta ve sıralı ifadeler çoğaltılmaktadır. Döngü gövdesindeki ifadenin kopyalanmasına döngü açma faktörü denir. Şekil 2.10.'daki ifadenin döngü açma faktörü ikidir.

Döngü açma tekniği derleyici optimizasyonu olduğu için döngünün ne kadar ilerleyeceği derlenme zamanında bilinmelidir. CUDA küçük döngüleri otomatik olarak optimize etmektedir. Uygulamamızda ise "#pragma unroll" talimatı ile en dıştaki döngü üzerinde döngü açma tekniği uygulanmıştır.

Döngü açma tekniği Bütün-çiftler, Karşılıklı kuvvetler ve Blokları düzenlemiş karşılıklı kuvvetler algoritmalarının üzerinde uygulanmış ve bunun çalışma zamanına (runtime) nasıl etki ettiği incelenmiştir.

```
for (i=0; i < 100; i++){
    x();
}

/////////////////////////

for (i=0; i < 50; i++){
    x();
    x();
}
```

Şekil 2.10. Faktörü 2 olan döngü açma

3. BULGULAR VE TARTIŞMA

3.1. Giriş

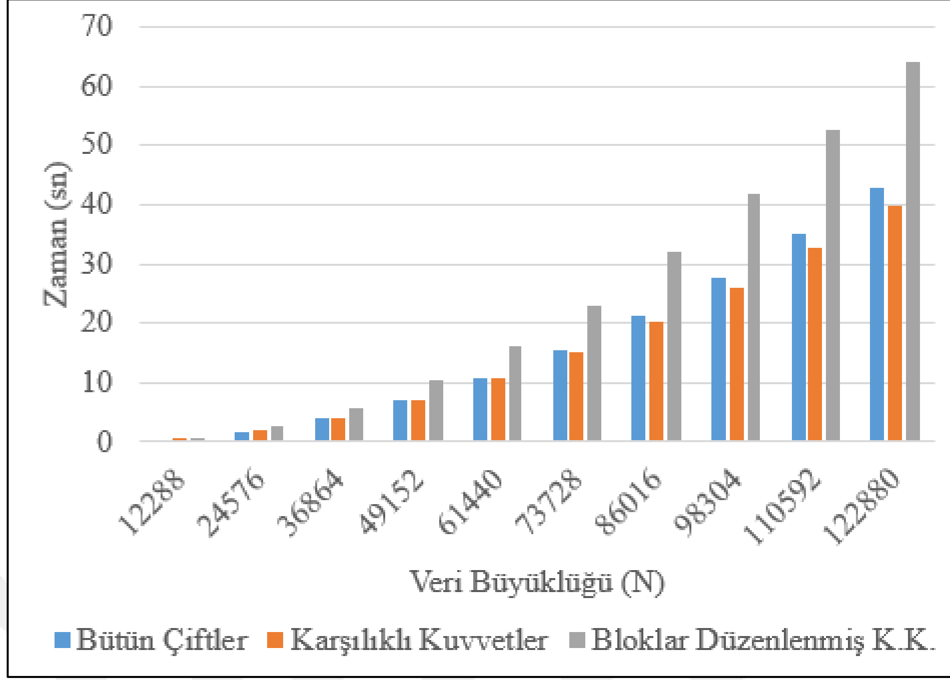
Tez kapsamında yapılan çalışmalar, Nvidia Nsight Eclipse ortamında kodlanarak, Nvidia Görsel Profil (nvvp) ve Nvidia Komut Satırı Profil (nvprof) ile test ve analiz edilmiştir. İlgili çalışmalar, Ubuntu 16.04 işletim sisteminde CUDA 9.1 araç takımı (toolkit) kullanılarak, GeForce GTX 760 (192 bit) ekran kartı üzerinde uygulanmıştır. GeForce GTX 760 (192 bit) ekran kartının teorik sınır değerleri, hesaplama gücü olarak 2.047 Tera FLOP/s ve bellek bant genişliği olarak da 134.4 GB/s'dir [66].

Yapılan çalışmada test verisi olarak, Samanyolu ve Andromeda galaksi çarpışmasının başlangıç parçacık pozisyonlarını içeren Dubinksi 1995 [67, 68] verisi kullanılmıştır. Bu veri 7 sütun (w,x,y,z,vx,vy,vz) 81920 satırdan oluşmaktadır. Sütunlar parçacıkların kütlesini(w), başlangıç konumlarını (x,y,z) ve başlangıç hızlarını (vx,vy,vz) temsil etmektedir.

3.2. Çalışma Zamanına Göre Algoritmaların Karşılaştırılması

Bu bölümde, Bütün-çiftler algoritması (BÇA), Karşılıklı Kuvvetler Optimizasyonu (KKO) ve Blokları Düzenlenmiş Karşılıklı Kuvvetler Optimizasyonu (BDKKO) çalışma zamanı bakımından karşılaştırılmıştır. Bu üç algoritmanın da karmaşıklığı $O(N^2)$'dir.

Şekil 3.1'de görüldüğü gibi ilk dört veride az farkla da olsa Bütün-çiftler algoritması daha iyi sonuç vermektedir. Daha sonra küçük farkla da olsa Karşılıklı kuvvetler algoritması daha iyi olmaya başlamaktadır. Dikkat çekici olan ise Blokları düzenlenmiş karşılıklı kuvvetler algoritması belirgin bir şekilde kötü sonuç vermesidir. Bunun sebebi her döngü adımında veri ulaşım deseninin değiştirilmesidir. Döngüdeki her adımda bloklar arası iş yükünü eşit dağıtmak adına veri ulaşım deseninin değiştirilmesi global bellek kullanımını çok olumsuz etkilemiştir. Bu durum Şekil 3.1'de açık bir şekilde görülebilir.



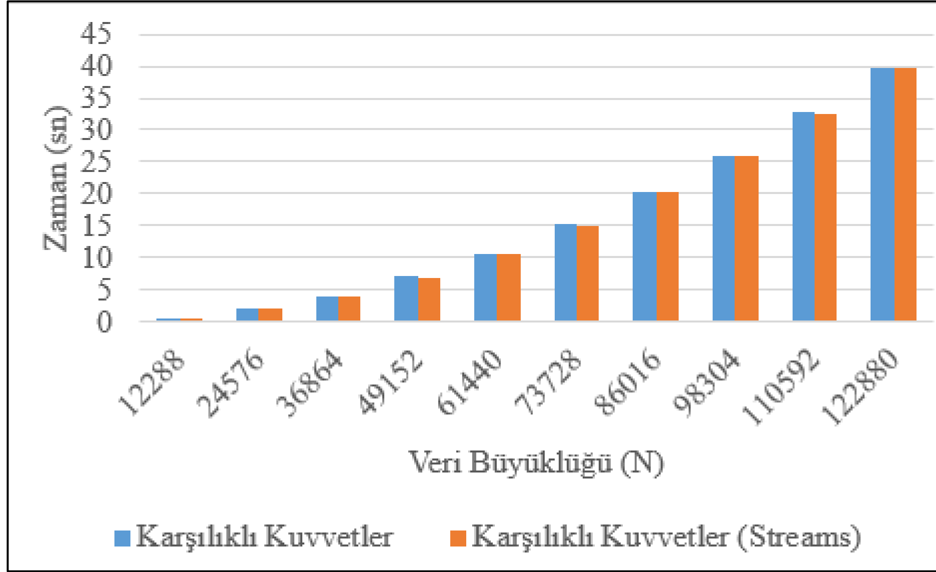
Şekil 3.1. Algoritmaların çalışma zamanı

3.3. Streams Özelliğinin Kullanımına Göre Algoritmalarının Karşılaştırılması

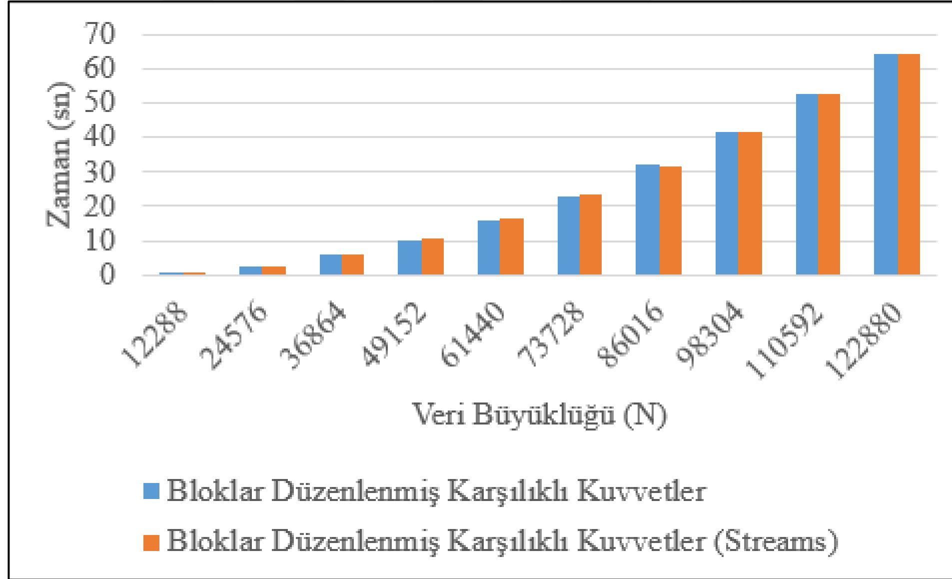
Bu bölümde Karşılıklı kuvvetler optimizasyonu ve Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonu hem Streams özelliği kullanılarak hem de kullanılmayarak uygulanmıştır. Streams kullanılması kodlamayı her ne kadar basitleştirse de Şekil 3.2'de görüldüğü gibi herhangi bir performans artışına neden olmamıştır. Bu yüzden bundan sonraki karşılaştırmalarda Streams kullanılan algoritmalar değerlendirmeye katılmamıştır.

3.4. FLOP/s Değerine Göre Algoritmaların Karşılaştırılması

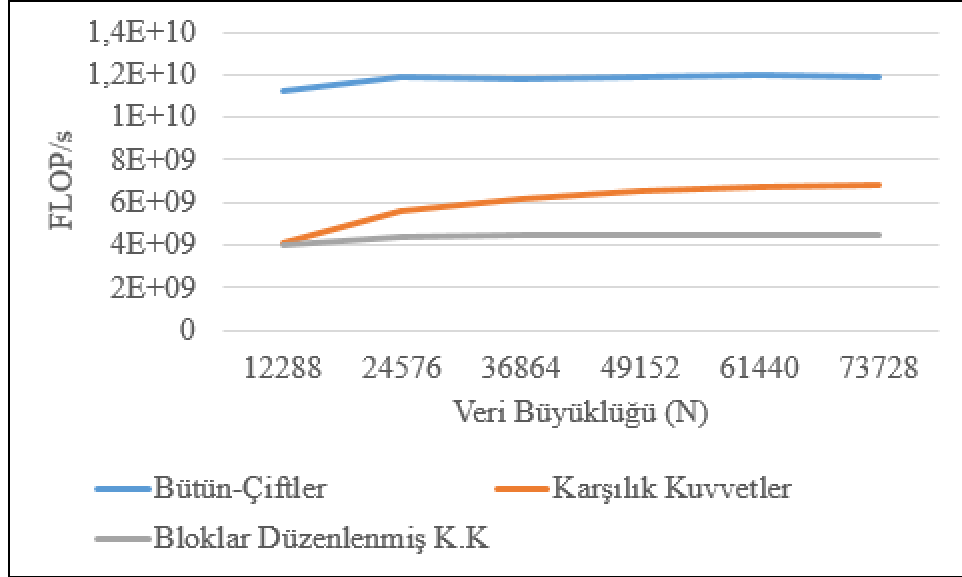
Paralel programlamada en önemli iki metrikten birincisi flop/s'dir (saniyedeki tek duyarlılıklı kayan noktalı sayı hesaplaması). Şekil 3.4.'de görüldüğü gibi Bütün-çiftler yöntemi Karşılıklı kuvvetler optimizasyonundan neredeyse 2 kat daha iyidir. Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonundan ise neredeyse 3 kat daha iyidir. Hesaplanan toplam işlem sayısı ise Şekil 3.5.'de gösterilmiştir.



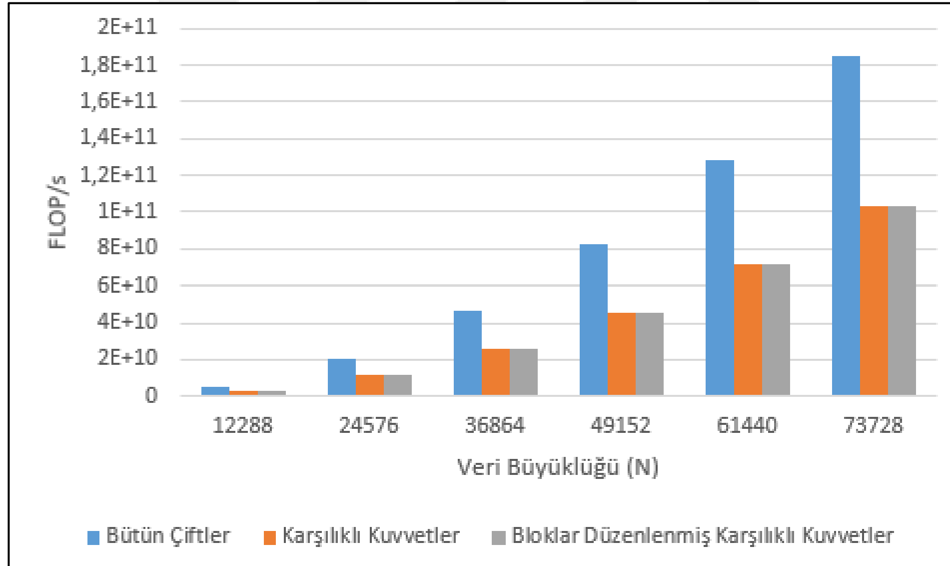
Şekil 3.2. Streams kullanımına göre KKO



Şekil 3.3. Streams kullanımına göre BDKKO



Şekil 3.4. Algoritmaların FLOP/s karşılaştırması

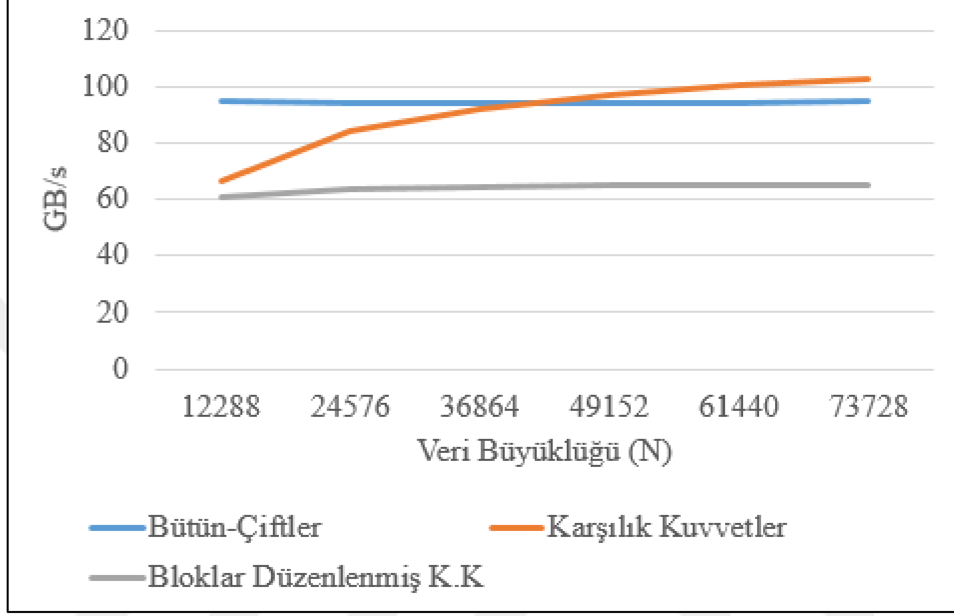


Şekil 3.5. Algoritmaların FLOP karşılaştırması

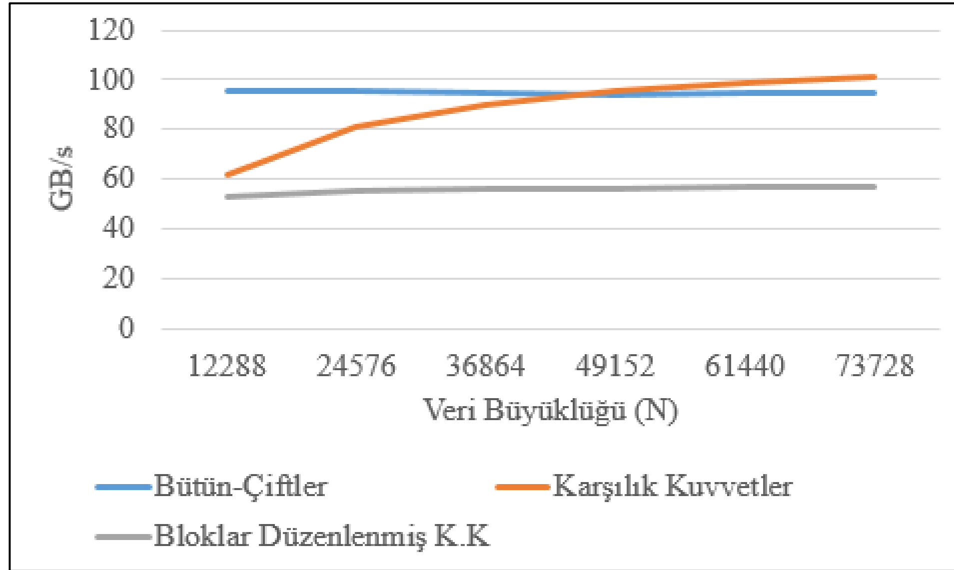
3.5. Bellek Kullanımlarına Göre Algoritmaların Karşılaştırılması

Paralel programlamada en önemli iki metrikten ikincisi bellek kullanımıdır. Algoritmaların L1 önbellek kullanımı Şekil 3.6.'de, ve L2 önbellek kullanımı Şekil 3.7.'de

gösterilmiştir. Karşılıklı kuvvetler optimizasyonu L1 ve L2 önbellekten en çok faydalanan algoritma olmuştur. Başlangıçta daha düşük seviyede faydalanmasının sebebi veri küçükken aktif thread sayısının az olmasıdır. Veri büyüdükçe aktif thread sayısı artığından önbellekten faydalanma durumu da artmaktadır. Bu durum bölüm 3.7 anlatılmıştır.

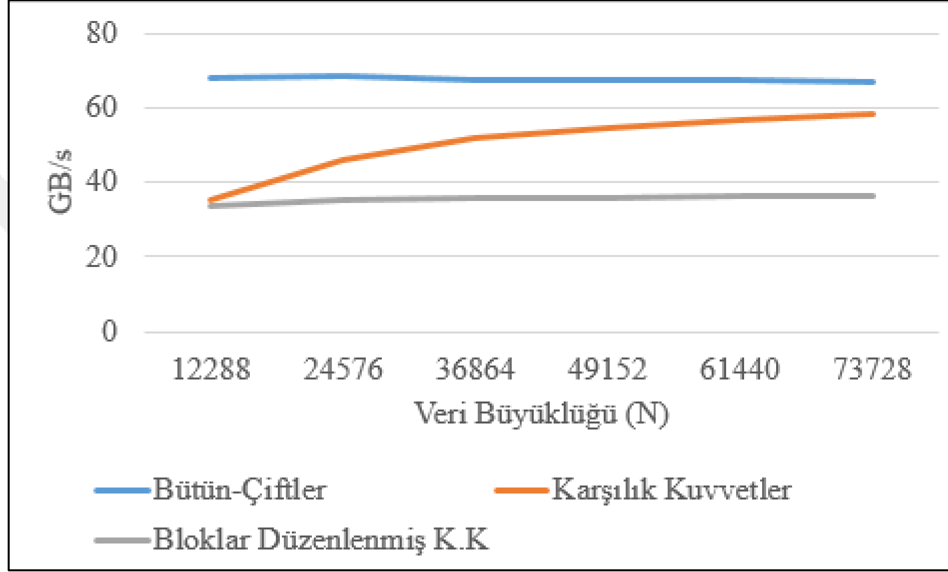


Şekil 3.6. Algoritmaların L1 önbellek kullanımı



Şekil 3.7. Algoritmaların L2 önbellek kullanımı

Şekil 3.8.'de algoritmaların global bellek kullanımı gösterilmiştir. Burada global bellekten en çok faydalanan algoritma Bütün-çiftler yöntemi olmuştur. Bunun en önemli sebebi bütün-çiftler yönteminin önbelleği yeteri kadar iyi kullanamamasıdır. Aynı şekilde karşılıklı kuvvetler yöntemi önbelleği daha iyi kullandığından global bellek kullanımı daha düşüktür. Burada da başlangıçta düşük bellek kullanımının sebebi aktif thread sayısının az olmasıdır.



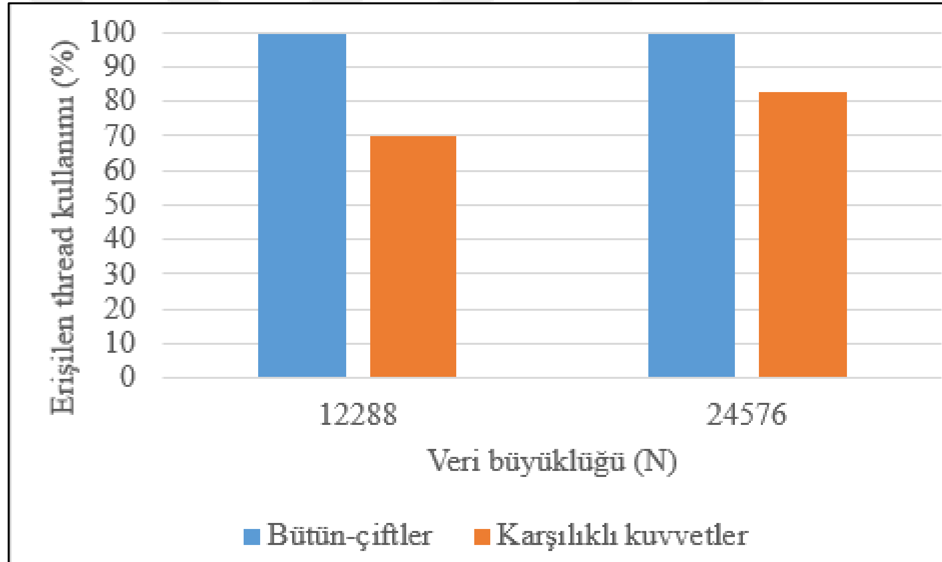
Şekil 3.8. Algoritmaların global bellek kullanımı

3.6. Erişilen Thread Kullanım Oranına Göre Algoritmaların Karşılaştırılması

Bu bölümde Bütün-çiftler ve Karşılıklı kuvvetler algoritmaları erişilen thread kullanım oranı (achieved occupancy) bakımından karşılaştırılmıştır. Fakat CUDA komut satırı profil (nvprof) aracı fazla işlem sayısından dolayı büyük veri uzunluğu için gereken bilgileri toplayamamıştır. Sadece ilk iki veri için bu değerler elde edilmiştir. Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonu tutarlı sonuçlar vermediğinden bu bölümde değerlendirmeye alınmamıştır.

Şekil 3.9.'da görüldüğü gibi Bütün-çiftler algoritmasında erişilen thread kullanım oranı (achieved occupancy) neredeyse %100'dür. Bu da algoritmanın basitliği ile birleşince Şekil 3.1.'de başlangıçta neden daha başarılı olduğunu açıklamaktadır.

Karşılıklı kuvvetler algoritmasının Şekil 2.2.'deki yapısından dolayı thread kullanım oranı veri küçükken düşük kalmaktadır. Örneğin veri uzunluğu 12288 iken 12288'inci thread sadece bir işlem yapmıştır. Bu da Şekil 3.9.'da görüldüğü gibi Karşılıklı kuvvetler optimizasyonunun thread kullanım oranını düşürmektedir. Fakat veri uzunluğu büyüdükçe bu oran yükselmeye başlamaktadır. Veri uzunluğu 2 katına çıktığında Karşılıklı kuvvetler optimizasyonun thread kullanım oranı %70'lerden %82'ye çıkmıştır. Bu sonuç da Karşılıklı kuvvetler optimizasyonunun Şekil 3.1.'de çalışma zamanının, Şekil 3.6.'da L1 önbellek kullanımının, Şekil 3.7.'de L2 önbellek kullanımının ve Şekil 3.8.'de global bellek kullanımının neden başlangıçta daha düşük performans sergilediğini göstermektedir. Ama veri uzunluğu her ne kadar artsa da Karşılıklı kuvvetler optimizasyonunun Şekil 2.2.'deki yapısından ötürü Bütün-çiftler algoritması gibi %100 performansa ulaşamayacaktır.

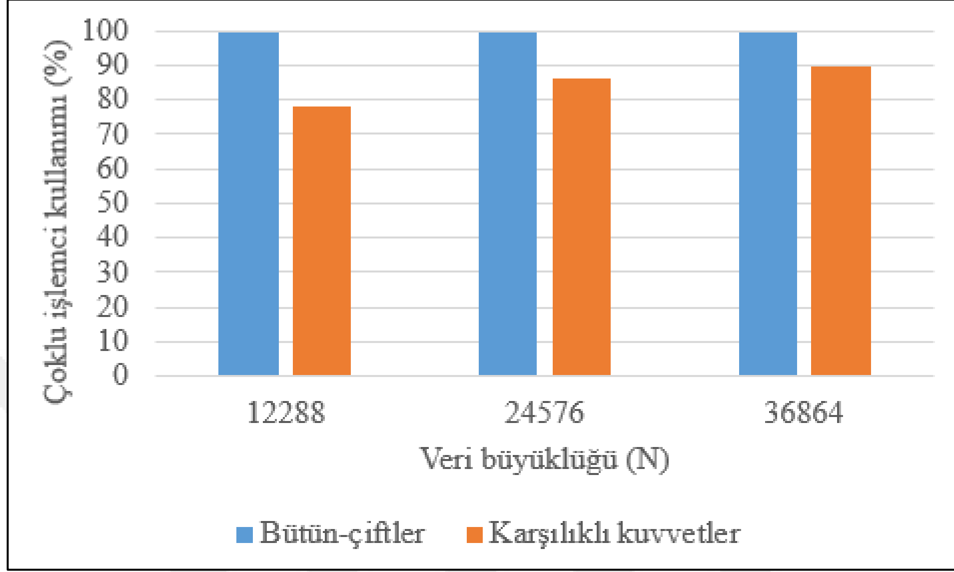


Şekil 3.9. Algoritmaların erişilen thread kullanım oranı

3.7. SM Kullanım Oranına Göre Algoritmaların Karşılaştırılması

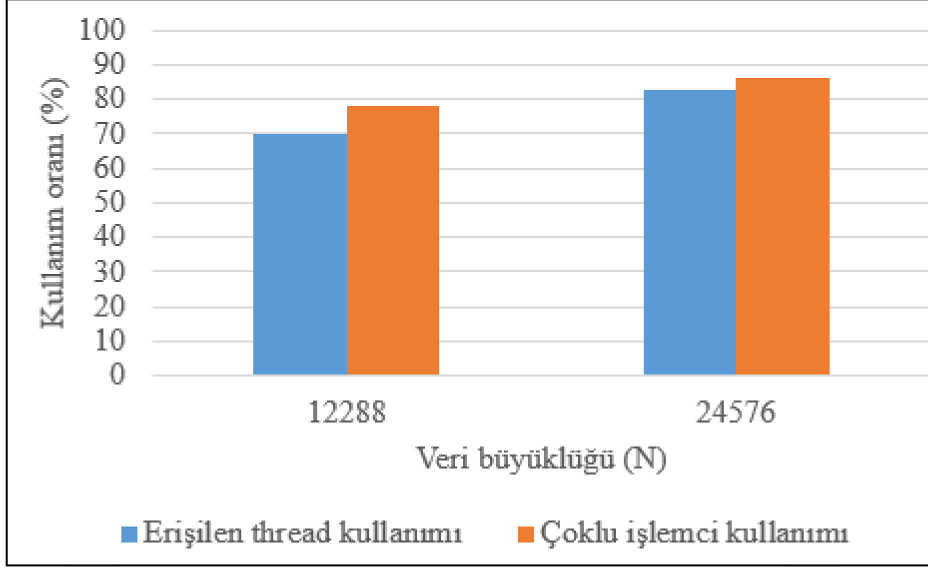
Bu bölümde, Bütün-çiftler ve Karşılıklı kuvvetler algoritmaları SM kullanım oranı açısından karşılaştırılmıştır. Fakat burada da CUDA komut satırı profil (nvprof) aracı fazla işlem sayısından dolayı büyük veri uzunluğu için gereken bilgileri toplayamamıştır. Sadece ilk üç veri için bu değerler elde edilmiştir. Blokları düzenlenmiş karşılı kuvvetler optimizasyonu tutarlı değerler vermediği için bu bölümde de değerlendirilmemiştir.

Şekil 3.10.’da görüldüğü gibi bütün-çiftler yöntemi erişilen thread kullanım oranında (achieved occupancy) olduğu gibi SM kullanımında da neredeyse %100’lük bir başarı elde etmiştir.



Şekil 3.10. Algoritmaların çoklu işlemci kullanımı

Karşılıklı kuvvetler optimizasyonu Şekil 3.10.’da görüldüğü gibi başlangıçta yine kötü performans göstermektedir. Bunun sebebi bölüm 3.6’da anlatılmaktadır. Karşılıklı kuvvetler optimizasyonunun SM kullanımı ve erişilen thread kullanımı (achieved occupancy) karşılaştırılması Şekil 3.11.’de gösterilmiştir. Bu iki değer birbirine benzer olsa da tam olarak aynı değildir. Birincisi problem için ayrılan kaynakların ne kadar verimli kullanıldığını ifade ederken, ikincisi problem için ayrılan thread’lerin ne kadar verimli kullanıldığını göstermektedir. Bu iki değer arasındaki farkı bölüm 3.6’da anlatılan sebepten dolayı olmaktadır. Yine aynı sebeplerden ötürü veri uzunluğu artsa bile bu iki değer birbirine yaklaşacak ama eşit olmayacaktır.



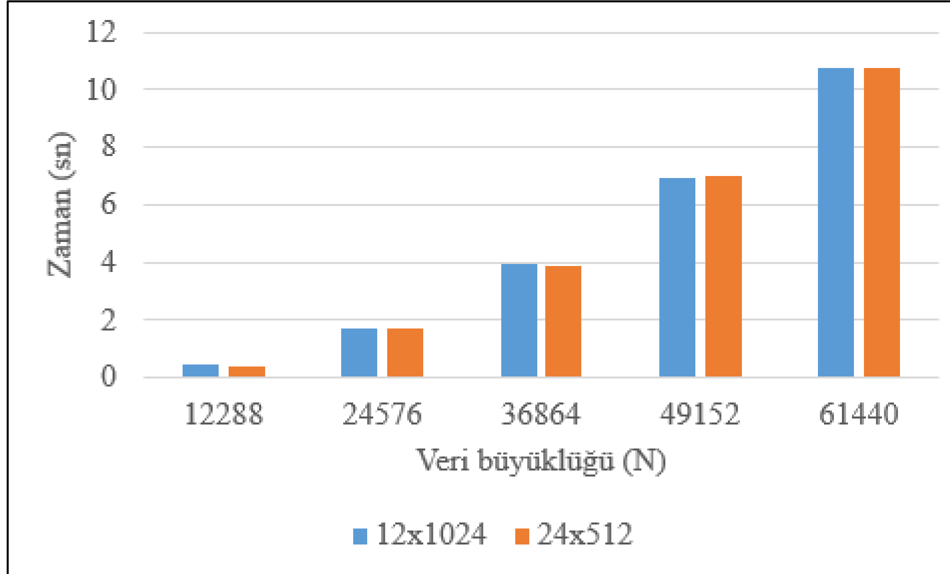
Şekil 3.11. KKO'nun erişilen thread ve çoklu işlemci kullanımı

3.8. Farklı Grid ve Blok Yapılarına Göre Algoritmalarının Karşılaştırılması

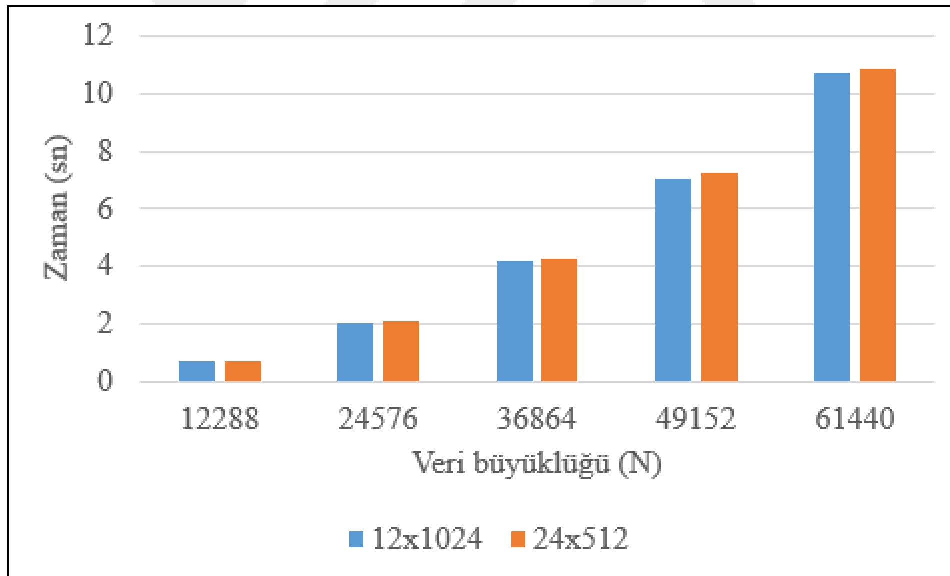
Bu bölümde, Bütün-çiftler, Karşılıklı kuvvetler ve Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonlarının farklı thread organizasyonu ile denenmiş ve çalışma zamanına karşılaştırmıştır. Şekil 3.12., Şekil 3.13. ve Şekil 3.14.'de görüldüğü gibi hiçbir algoritmada kayda değer bir gelişme olmamıştır. Bunun en önemli sebebi bütün thread'lerin aktif olarak kullanılmasıdır. Daha düşük thread kullanımında farklı sonuçlar ortaya çıkması daha olasıdır.

3.9. Paylaşımlı Bellek Kullanımına Göre BÇA'nın Karşılaştırılması

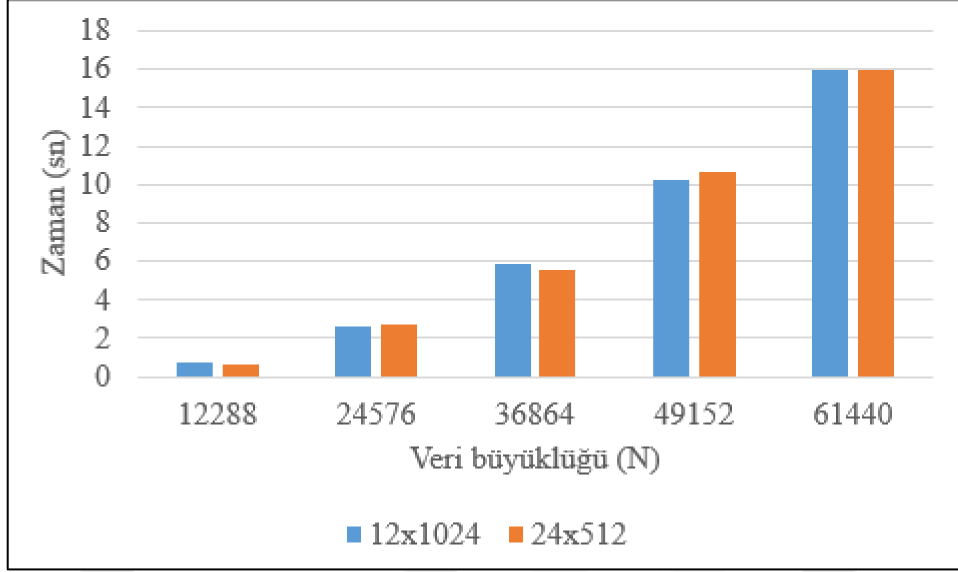
Bu bölümde, Bütün-çiftler yöntemi paylaşımlı bellek kullanılarak uygulanmış ve paylaşımlı bellek kullanımına göre çalışma zamanı açısından karşılaştırılmıştır. Şekil 3.15.'deki gibi paylaşımlı bellek kullanıldığında çok küçük bir iyileşme olmuştur. Bunun sebebi, L1 önbelleği açıkça kullanılmasa da Şekil 3.16.'da görüldüğü gibi CUDA derleyicisi kodu optimize ederek L1 önbelleğini aktif olarak kullanmıştır. İki uygulama arasında Bundan dolayı iki uygulama arasında önemli bir farklılık olmamıştır.



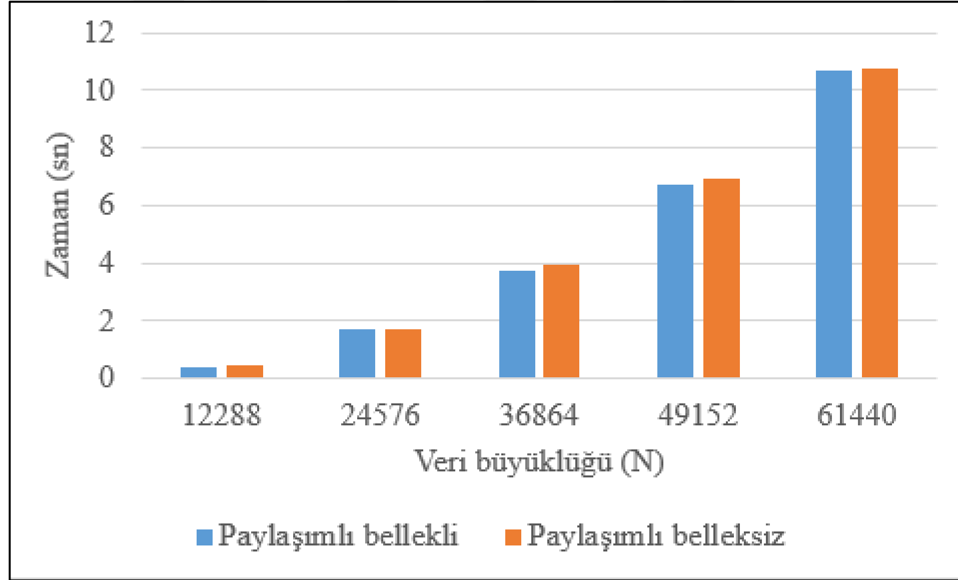
Şekil 3.12. BÇA'nın farklı grid ve blok yapısında karşılaştırılması



Şekil 3.13. KKO'nun farklı grid ve blok yapısında karşılaştırılması



Şekil 3.14. BDKKO'nun farklı grid ve blok yapısında karşılaştırılması

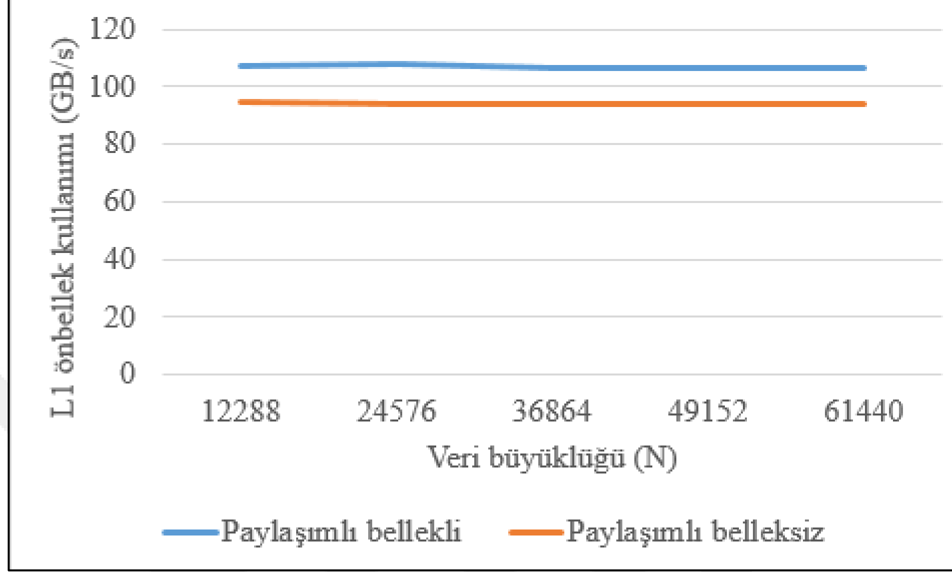


Şekil 3.15. BÇA'nın paylaşımlı bellek kullanımı

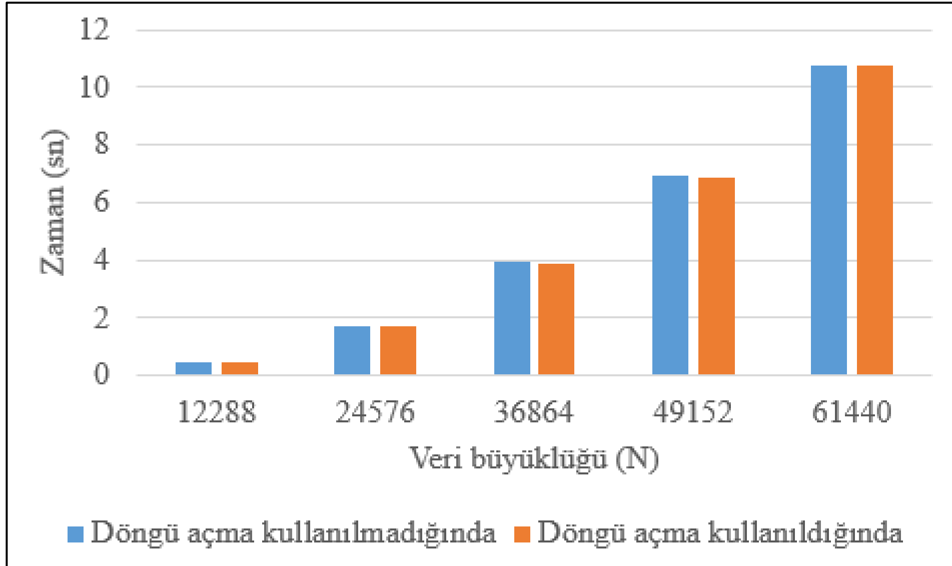
3.10. Döngü Açma Optimizasyonuna Göre Algoritmalarının Karşılaştırılması

Bu bölümde, Bütün-çiftler, Karşılıklı kuvvetler ve Blokları düzenlenmiş karşılıklı kuvvetler algoritmalarına döngü açma tekniği uygulanmıştır. Şekil 3.17.'de görüldüğü gibi Bütün-çiftle yönteminde saniyeden daha küçük iyileştirmeler olmuştur. Şekil 3.18. ve Şekil

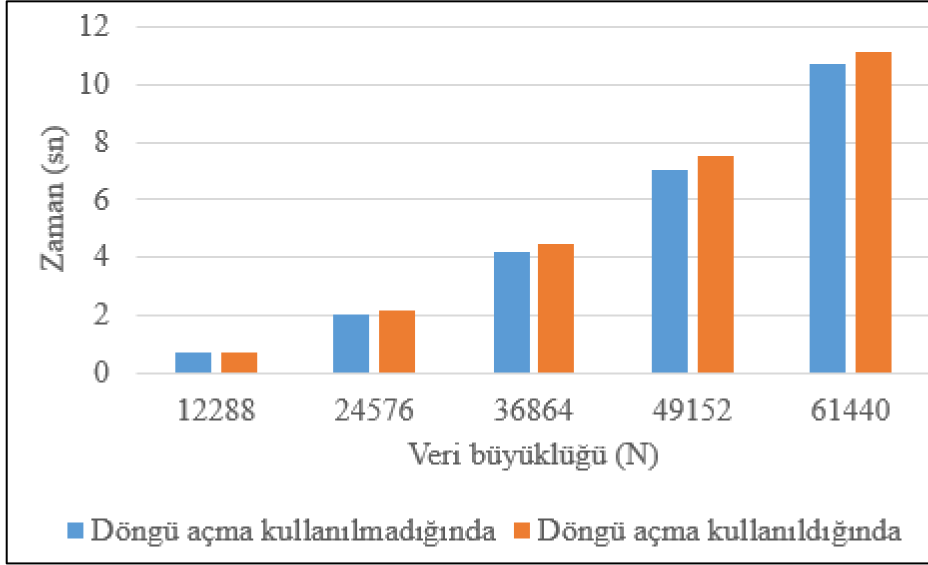
3.19.’da görüldüğü gibi Karşılıklı kuvvetler ve Blokları düzenlenmiş karşılıklı kuvvetler algoritmalarında saniyeden daha küçük kötüleşme olmuştur. Fakat iyileştirme ve kötüleştirmeler kayda değer ölçüde değildir.



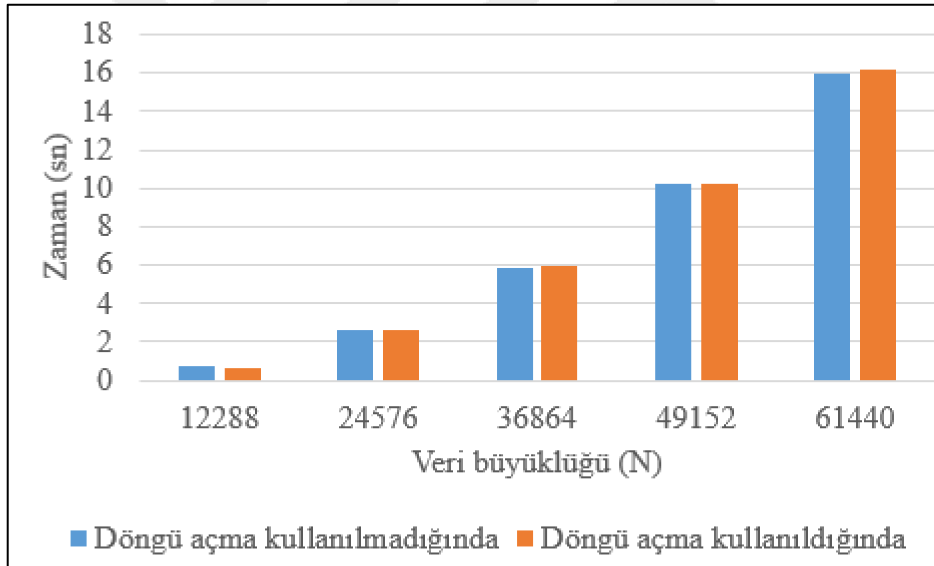
Şekil 3.16. BÇA’nın paylaşımlı bellek kullanımın L1 önbelleğe etkisi



Şekil 3.17. BÇA’nda döngü açma yöntemi



Şekil 3.18. KKO'nunda döngü açma yöntemi



Şekil 3.19. BDKKO'nunda döngü açma tekniği

4. SONUÇLAR

Bu tez çalışmasında, daha önce literatürde analiz edilmemiş olan yerçekimsel N-cisim problemindeki Karşılıklı kuvvetler optimizasyonu GPU üzerinde CUDA ile analiz edilmiştir. Karşılıklı kuvvetler optimizasyonu, CUDA Streams özelliği kullanımına göre iki farklı ve yeni algoritma ile uygulanmıştır. Analizler N-cisim problemindeki Bütün-çiftler yöntemi ile karşılaştırılarak yapılmıştır. CUDA Streams özelliği kullanılarak uygulanan Karşılıklı kuvvetler optimizasyonu, 5. İleri Teknoloji ve Bilimler Uluslararası Konferansı (ICAT'17) adlı konferansta 1 adet bildiri olarak [69], sözlü sunumu yapılmış ve bildiri kitapçığına basılmıştır.

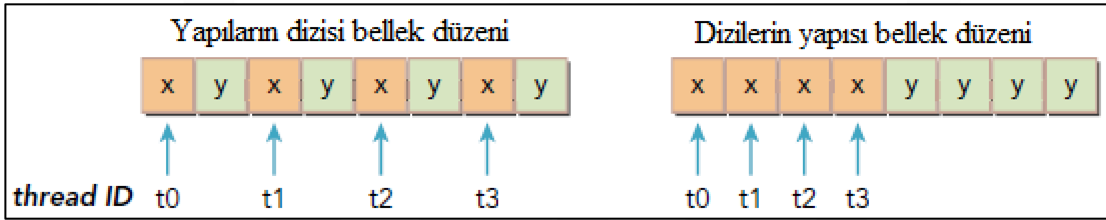
Analizler sonucu elde edilen bulgular aşağıda sıralanmıştır:

1. Veri uzunluğu küçükken, Karşılıklı kuvvetler optimizasyonunun yapısı gereği Bütün-çiftler yöntemine göre yürütme zamanı, hesaplama gücü (flop/s) ve bellek kullanımı gibi temel kriterlerde geride kaldığı gözlemlenmiştir.
2. Veri uzunluğu arttıkça, Karşılıklı kuvvetler optimizasyonu Bütün-çiftler yöntemine yürütme zamanı açısından üstünlük sağlamıştır.
3. Veri uzunluğu arttıkça, Karşılıklı kuvvetler optimizasyonu Bütün-çiftler yöntemine bellek kullanımı açısından denklik sağlamıştır.
4. Veri uzunluğu arttıkça, Karşılıklı kuvvetler optimizasyonu algoritmasının karmaşık oluşundan ve Bütün-çiftler algoritmasının basitliğinden ötürü hesaplama gücü (flop/s) kriterinde Karşılıklı kuvvetler optimizasyonu hiçbir şekilde üstünlük sağlayamamıştır. Bu sonuç, GPU üzerinde paralel hesaplama yaparken algoritma basitliğinin ne kadar önemli olduğunu göstermektedir.

Sonuç olarak, veri sayısının fazla olduğu simülasyonlarda Karşılıklı kuvvetler optimizasyonunun Bütün-çiftler yöntemine tercih edilebilir olduğu ispatlanmıştır.

5. ÖNERİLER

Bu çalışmada kullanılan veri organizasyonu yapıların dizisidir (array of structures). Veri organizasyonu dizilerin yapısına (structure of arrays) çevrilerek analizler tekrar yapılabilir. Dizililerin yapısı (structure of arrays) veri modeli GPU programlamaya daha uygun olduğundan [4] önemli farklılıklar elde edilebilir. Yapıların dizisinin (array of structures) ve dizilerin yapısının (structure of arrays) bellek düzeni Şekil 5.1.'de gösterilmiştir.



Şekil 5.1. Yapıların dizisi ve dizilerin yapısı bellek düzeni

Ayrıca Blokları düzenlenmiş karşılıklı kuvvetler optimizasyonu için uygun bir veri ulaşım deseni sağlanabilirse yürütme zamanında, hesaplama gücünde ve bellek kullanımında büyük oranda iyileştirme sağlanabilir. Ek olarak küçük verilerde bile Bütün-çiftler yöntemi ile yarışır hale gelebilir.

6. KAYNAKLAR

1. Dowd, K. ve Severance, C., High Performance Computing (RISC Architectures, Optimization & Benchmarks), Second Edition, O'Reilly Media, 1998.

2. Sterling, T., Anderson, M. ve Brodowicz, M, High Performance Computing: Modern Systems and Practices, First Edition, Morgan Kaufmann, 2017.

3. Gottlieb, A. ve Almasi, G. S., Highly Parallel Computing, First Edition, Benjamin-Cummings Publishing, 1989.

4. Cheng, J., Grossman, M. ve McKercher, T., Professional CUDA C Programming, First Edition, Wrox, 2014.

5. Hennessy, J. L. ve Patterson, D. A., Computer Architecture: A Quantitative Approach, Sixth Edition, Morgan Kaufmann, 2017.

6. Khaldi, D., Jouvelot, P. ve Ancourt, C. ve Irigoin, F., Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages, Languages and Compilers for Parallel Computing, Eylül 2012, Tokyo, Bildiriler Kitabı: 174 - 189.

7. Quinn, M., Parallel Programming in C with MPI and OpenMP, First Edition, McGraw-Hill , 2003.

8. Flynn, M. J., Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 21,9 (1972) 948-960.

9. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf NVIDIA Fermi Compute Architecture Whitepaper 21 Mayıs 2018.

10. Gao, Y. ve Zhang, P., A Survey of Homogeneous and Heterogeneous System Architectures in High Performance Computing, IEEE International Conference on Smart Cloud, Kasım 2016, New York, Bildiriler Kitabı: 170-175.

11. https://www.linuxjournal.com/article/8368 Heterogeneous Processing: a Strategy for Augmenting Moore's Law 21 Mayıs 2018.

12. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf Heterogeneous System Architecture: A Technical Review 21 Mayıs 2018.

13. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#heterogeneous-programming Heterogeneous Programming 21 Mayıs 2018.

14. https://developer.nvidia.com/cuda-zone CUDA Zone 21 Mayıs 2018.

15. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html CUDA C Programming Guide 21 Mayıs 2018.

16. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#driver-api Driver API 21 Mayıs 2018.

17. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-c-runtime CUDA C Runtime 21 Mayıs 2018.

18. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc Compilation with NVCC 21 Mayıs 2018.

19. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels Kernels 21 Mayıs 2018.

20. https://docs.nvidia.com/cuda/index.html CUDA Toolkit Documentation v9.2.88 21 Mayıs 2018.

21. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy Thread Hierarchy 21 Mayıs 2018.

22. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture SIMT Architecture 21 Mayıs 2018.

23. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy Memory Hierarchy 21 Mayıs 2018.

24. Denning P. J., The Locality Principle, Communications of the ACM, 48,7 (2005), 19-24.

25. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory Shared Memory 21 Mayıs 2018.

26. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory Device Memory 21 Mayıs 2018.

27. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams Streams 21 Mayıs 2018.

28. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution Asynchronous Concurrent Execution 21 Mayıs 2018.

29. http://developer.download.nvidia.com/compute/cuda/7_0/Prod/doc/CUDA_Toolkit_Release_Notes.pdf NVIDIA CUDA TOOLKIT V7.0 March 2015 Release Notes for Windows, Linux, and Mac OS 21 Mayıs 2018.

30. https://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/index.html Nsight Eclipse Edition Getting Started Guide 21 Mayıs 2018.

31. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual Visual Profiler 21 Mayıs 2018.

32. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview nvprof 21 Mayıs 2018.

33. Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T. ve Umemura, M., A Special-Purpose Computer for Gravitational Many-Body Problems, Nature, 345 (1990) 33-35.

34. Makino, J. ve Taiji, M., Scientific Simulations with Special-Purpose Computers - The GRAPE Systems, First Edition, John Wiley & Sons, New York, 1998.

35. Fukushige, T., Taiji, M., Makino, J., Ebisuzaki, T. ve Sugimoto, D., A Highly Parallelized Special-Purpose Computer for Many-Body Simulations with an Arbitrary Central Force: MD-GRAPE, Astrophysical Journal, 468 (1996) 51-61.

36. Susukita, R., Ebisuzaki, T., Elmegreen, B. G., Furusawa, H., Kato, K., Kawai, A., Kobayashi, Y., Koishi, T., Mc-Niven, G. D., Narumi, T. ve Yasuoka, K., Hardware accelerator for molecular dynamics: MDGRAPE-2, Computer Physics Communications, 155 (2003) 115-131.

37. Gingold, R. A., ve Monaghan, J. J., Smoothed particle hydrodynamics - Theory and application to non-spherical stars, Monthly Notices of the Royal Astronomical Society, 181 (1977) 375-389.

38. Monaghan, J. J., Smoothed particle hydrodynamics, Annual review of astronomy and astrophysics, 30 (1992) 543-574.

39. Brebbia, C. A., The Boundary Element Method for Engineers, First Edition, Pentech Press, London, 1978.

40. Hamada, T., Nitadori, K., Benkrid, K., Ohno, Y., Morimoto, G., Masada, T., A novel multiple-walk parallel algorithm for the Barnes-Hut treecode on GPUs - towards cost effective high performance N-body simulation, Computer Science - Research and Development, 24 1-2 (2009) 21-31.

41. Ishiyama, T., Nitadori, K. ve Makino, J., 4.45 pflops Astrophysical N-body Simulation on k Computer:The gravitational trillion-body problem, International Conference on High Performance Computing, Networking, Storage and Analysis, Kasım 2012, Salt Lake City-Utah, Bildiriler Kitabı: 1-10.

42. Warren, M. S., ve Salmon, J. K., Astrophysical N-body simulations using hierarchical tree data structures, ACM/IEEE conference on Supercomputing, Kasım 1992, Minneapolis-Minnesota, Bildiriler Kitabı: 570-576.

43. Fukushige, T. ve Makino, J., N-body Simulation of Galaxy Formation on GRAPE-4 Special-purpose Computer, ACM/IEEE conference on Supercomputing , Ocak 1996, Pittsburgh-Pennsylvania, Bildiriler Kitabı: 432-446.

44. Warren, M. S., Salmon, J. K., Becker, D. J., Goda, M. P., Sterling, T., ve Winckelmans, G. S., PentiumPro inside: I. a treecode at 430 Gflops on ASCI red, II. Price/performance of $50/Mflop on Loki and Hyglac, ACM/IEEE conference on Supercomputing, Kasım 1997, San Jose-CA, Bildiriler Kitabı: 61-61

45. Kawai, A., Fukushige, T., ve Makino, J., $7.0/mflops Astrophysical N-body Simulation with Treecode on Grape-5, ACM/IEEE Conference on Supercomputing, Kasım 1999, Portland-Oregon, Bildiriler Kitabı: 67-67.

46. Dyer, C. ve Ip, P., Softening in N-Body Simulations of Collisionless Systems, The Astrophysical Journal, 409 (1993) 60-67.

47. Aarseth, S. Gravitational N-Body Simulations: Tools and Algorithms, First Edition, Cambridge University Press, 2003.

48. Euler, L., Institutionum calculi integralis, First Edition, Lipsiae Et Berolini, 1768.

49. Nguyen, H., GPU Gems 3, First Edition, Addison-Wesley Professional, 2007.

50. Yokota, R. ve Lorena, B., Hierarchical N-body Simulations with Autotuning for Heterogeneous Systems, Computing in Science & Engineering, 14,3 (2012) 30 - 39.

51. Barnes, J. ve Hut, P., A hierarchical O (NlogN) force calculation algorithm, Nature, 324 (1986) 446-449.

52. Greengard, L. ve Rokhlin,V., A fast algorithm for particle simulations, Journal of Computational Physics, 73,2 (1987) 325-348.

53. Hockney, R.W. ve Eastwood J.W, Computer Simulation Using Particles, First Edition, CRC Press, 1988.

54. Volker, S., The cosmological simulation code GADGET-2, Monthly Notices of the Royal Astronomical Society, 364,4 (2005) 1105-1134.

55. Ito, T., Makino, J., Ebisuzaki, T. ve Sugimoto, D., A special-purpose N-body machine GRAPE-1, Computer Physics Communications, 60,2 (1990) 187-194.

56. Makino, J. ve Daisaka, H., GRAPE-8 -- An accelerator for gravitational N-body simulation with 20.5Gflops/W performance, International Conference for High Performance Computing, Networking, Storage and Analysis, Kasım 2012, Washington-DC, Bildiriler Kitabı: 1-10.

57. Jiang, H. ve Deng, Q., Barnes-hut treecode on GPU, 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), Kasım 2010, Shanghai, Bildiriler Kitabı: 974-978.

58. Yokota, R. ve Barba, L., Hierarchical N-body Simulations with Autotuning for Heterogeneous Systems, Computing in Science & Engineering, 14,3 (2012) 30-39.

59. Siegel, J., Ributzka, J. ve Li. X., CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator, 2009 International Conference on Parallel Processing Workshops, Eylül 2009, Vienna, Bildiriler Kitabı: 174-181.

60. Camillo. M.S. ve Shin-Ting, W., Accessing CUDA Features in the OpenGL Rendering Pipeline: A Case Study Using N-Body Simulation, 30th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI), Ocak 2017, Niteroi, Bildiriler Kitabı: 315-322.

61. Aquotte, F. A. ve Silva, A. F., PSIM: A Modular Particle System on Graphics Processing Unit, IEEE Latin America Transactions, Ocak 2014, Brazil, Bildiriler Kitabı: 321-329.

62. Hu, Q., Gumerov, N. A. ve Duraiswami, R., Scalable fast multipole methods on distributed heterogeneous architectures, International Conference for High Performance Computing, Networking, Storage and Analysis, Kasım 2011, Seatle, Bildiriler Kitabı: 1-12.

63. Gaburov, E., Harfst, S. ve Zwart, S.P., Sapporo: A Way to Turn Your Graphics Cards into a GRAPE-6, New Astronomy, 14,7 (2009) 630-637.

64. https://wwwmpa.mpa-garching.mpg.de/mpa/research/current_research/hl2011-9/hl2011-9-en.html The Millennium-XXL Project: Simulating the Galaxy Population in Dark Energy Universes 21 Mayıs 2018.

65. https://wwwmpa.mpa-garching.mpg.de/gadget/ GADGET-2: A code for cosmological simulations of structure formation 21 Mayıs 2018.

66. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-760-oem/specifications Specifications 21 Mayıs 2018.

67. http://bima.astro.umd.edu/nemo/archive/#dubinski Dubinksi 1995 21 Mayıs 2018.

68. Dubinski, J., Mihos, J. C. ve Hernquist, L., Using Tidal Tails to Probe Dark Matter Halos, The Astrophysical Journal, 462 (1996) 576-603.

69. Özkurt, C. ve Gedikli E., 5th International Conference on Advanced Technology & Sciences, Mayıs 2017, İstanbul, Bildiriler Kitabı:326-329

ÖZGEÇMİŞ

Celil Özkurt, 1985 İstanbul doğumludur. 2009 yılında Fatih Üniversitesi Bilgisayar Mühendisliği Bölümü'nden bölüm birincisi olarak mezun olmuştur. İyi seviyede İngilizce bilmektedir.

- Ağustos 2009 – Mart 2010 tarihleri arasında Demiryollarında Makas Mekanizmaları için Arıza Öngörme ve Bakım Planlama Sisteminin Geliştirilmesi (Development of Failure Prognostics and Maintenance Planning System for Point Mechanisms in Railways) adlı TÜBİTAK projesinde bursiyer olarak çalışmıştır.

- Nisan 2010 – Nisan 2011 tarihleri arasında askerlik görevini Asteğmen olarak Ankara'da yapmıştır.

- Eylül 2012 – Ağustos 2012 arasında bir yıl özel sektör tecrübesi vardır.

- Eylül 2012 – Ekim 2013 arasında Kafkas Üniversitesinde araştırma görevlisi olarak çalışmıştır.

- Eylül 2013 – Ocak 2015 tarihleri arasına Batarya Yönetim Sistemlerinde Dizayn Bazlı Sağlık Durumu ve Kalan Faydalı Ömür Tespit Yöntemi Geliştirilmesi ve Şarj Edilebilir Bataryalara Uygulanması (Development of Design based State-of-Health and Remaining Useful Life Estimation Techniques for Battery Management Systems and Its Application to Rechargable Batteries) adlı TÜBİTAK projesinde bursiyer olarak çalışmıştır.

- Eylül 2014 - Eylül 2017 yılları arasında Karadeniz Teknik Üniversitesi Yazılım Mühendisliği Bölümünde araştırma görevlisi olarak çalışmıştır. Aynı tarihte Karadeniz Teknik Üniversitesi Bilgisayar Mühendisliği bölümünde yüksek lisansa başlamıştır.