ANKARA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

GENEL ATAMA PROBLEMLERİNİN GRAFİK İŞLEMCİ BİRİMLERİNİN ÜZERİNDE ÇÖZÜMÜ

Eren AKÇA

ELEKTRİK-ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

ANKARA
2015

TEZ ONAYI

Eren AKÇA tarafından hazırlanan "**Genel Atama Problemlerinin Grafik İşlemci Birimlerinin Üzerinde Çözümü**" adlı tez çalışması 23/10/2015 tarihinde aşağıdaki jüri tarafından oy birliği ile Ankara Üniversitesi Fen Bilimleri Enstitüsü Elektrik Elektronik Mühendisliği Ana Bilim Dalı'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Danışman: Doç. Dr. Asım Egemen YILMAZ

Jüri Üyeleri:

Başkan **:** Doç. Dr. S. Çağdaş İNAM
Başkent Üniversitesi
Elektrik-Elektronik Mühendisliği Bölümü

Üye **:** Doç. Dr. Asım Egemen YILMAZ
Ankara Üniversitesi
Elektrik-Elektronik Mühendisliği Anabilim Dalı

Üye **:** Yrd. Doç. Dr. Gökhan SOYSAL
Ankara Üniversitesi
Elektrik-Elektronik Mühendisliği Anabilim Dalı

Yukarıdaki sonucu onaylıyorum.

Prof. Dr. İbrahim DEMİR
Enstitü Müdürü

ETİK

Ankara Üniversitesi Fen Bilimleri Enstitüsü tez yazım kurallarına uygun olarak hazırladığım bu tez içindeki bütün bilgilerin doğru ve tam olduğunu, bilgilerin üretilmesi aşamasında bilimsel etiğe uygun davrandığımı, yararlandığım bütün kaynakları atıf yaparak belirttiğimi beyan ederim.

23 Ekim 2015

Eren AKÇA

ÖZET

Yüksek Lisans Tezi

GENEL ATAMA PROBLEMLERİNİN GRAFİK İŞLEMCİ BİRİMLERİNİN ÜZERİNDE ÇÖZÜMÜ

Eren AKÇA

Ankara Üniversitesi
Fen Bilimleri Enstitüsü
Elektrik-Elektronik Mühendisliği Anabilim Dalı

Danışman: Doç. Dr. Asım Egemen YILMAZ

Gelişen savunma teknolojilerine karşın kaynakların giderek tükendiği günümüzde, hava araçları görevlerini icra ederken minimum yakıt tüketimi, süre ve görev riski ile maksimum başarı elde etmek zorundadır. Kısıtlı sürede, her bir hava aracı için birden fazla amaç gözetilerek görevlerin planlanması zorunluluğu ortaya Çok Amaçlı Optimizasyon Problemini çıkarmaktadır. Tez kapsamında yapılan çalışmada hava araçlarının; istenilen görevleri en verimli şekilde yerine getirebilmeleri için, görev planlamalarında kullanılacak olan çok-amaçlı (multi-objective) optimizasyon algoritmaları geliştirilmiştir. Daha sonra bu algoritmalar, etkin bir şekilde Genel Amaçlı Grafik İşlemci Birimleri (General Purpose Graphics Processing Units - GPGPUs) üzerinde CUDA programlama dili ile paralelleştirilerek bir yazılım arayüzü oluşturulmuştur. Çalışmadan elde edilen sonuçlar, çok amaçlı optimizasyon problemlerinin oluşturulan yazılım arayüzü ile çözülmesinin, merkezi işlemci birimi üzerinde seri olarak çözülmesine oranla 16 kata kadar daha hızlı olduğu göstermiştir.

Ekim 2015, 76 sayfa

Anahtar Kelimeler: Çok amaçlı optimizasyon, görev planlama, genetik algoritma, Grafik İşlemci Birimleri, GPGPU, CUDA, skalarizasyon teknikleri

ABSTRACT

Master Thesis

SOLUTION OF THE GENERALIZED ASSIGNMENT PROBLEMS ON GRAPHICS PROCESSING UNITS

Eren AKÇA

Ankara University
Graduate School of Natural and Applied Sciences
Department of Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Asım Egemen YILMAZ

Nowadays increasingly depleted of resources despite the developing defense technologies, air vehicles must achieve success while conducting their mission with minimum fuel consumption, time and mission risk. In limited time, the obligation of planning mission for each air vehicle by taking into account more than one objective reveals a multi-objective optimization problem. In the thesis work, multi-objective optimization algorithms used in mission planning of the air vehicles is developed in order to fulfill the desired mission. After that, a software interface is created by these algorithms parallelized on General Purpose Graphics Processing Units (GPGPUs) via CUDA programming language in an efficient manner. The results obtained from the study show that solving the multi-objective optimization problems via the created software interface is up to 16 times faster than solving them in a serial manner on the central processing unit.

October 2015, 76 pages

Key Words: Multi-objective optimization, mission planning, genetic algorithm, Graphics Processing Units, GPGPU, CUDA, scalarization techniques

TEŞEKKÜR

Tez çalışması süresince bana engin bilgileri ile yol gösterip yardımını esirgemeyen ve yüksek lisans eğitimim boyunca kendisinden çok şey kazandığım çok değerli danışman hocam Doç. Dr. Asım Egemen YILMAZ (Ankara Üniversitesi Fen Bilimleri Enstitüsü Elektrik-Elektronik Mühendisliği Anabilim Dalı)'a ve değerli fikirleri ile çalışmamıza katkıda bulunan hocam Doç. Dr. Murat EFE (Ankara Üniversitesi Fen Bilimleri Enstitüsü Elektrik-Elektronik Mühendisliği Anabilim Dalı)'ye teşekkür ederim.

Tez çalışması boyunca mükemmel bir ekip olduğumuzu hissettiren ve kendileri ile birlikte çalışmaktan gurur duyduğum başta Tayfur YAYLAGÜL ve Sadi Uçkun EMEL ile diğer çalışma arkadaşlarıma teşekkür ederim.

Ayrıca yüksek lisans eğitimim boyunca yaptığım çalışmayı 01568.STZ.2012-2 numaralı ve "Grafik İşlemci Birimleri Üzerinde Genel Atama Problemlerinin Çözümü" başlıklı SANTEZ projesi kapsamında destekleyen Bilim, Sanayi ve Teknoloji Bakanlığı ile HAVELSAN A.Ş'ye teşekkür ederim.

Son olarak, desteklerini hiç bir zaman benden esirgemeyen, her zaman yanımda olan sevgili eşim Zülbiye AKÇA'ya ve aileme teşekkürü bir borç bilirim.

Eren AKÇA
Ankara, Ekim 2015

İÇİNDEKİLER

SİMGELER DİZİNİ

GB	Gigabayt
GHz	Gigahertz
MHz	Megahertz
S	Saniye

Kısaltmalar

ACO	Karınca Kolonisi Optimizasyonu
ALU	Aritmetik Mantık Birimi
API	Uygulama Programlama Arayüzü
BIOMA 2014	Bioinspired Optimization Methods and their Application 2014 Workshop
CPU	Merkezi İşlemci Birimi
CUDA	İşlemsel Birleştirilmiş Cihaz Mimarisi
DRAM	Dinamik Rastgele Erişimli Hafıza
FPGA	Alan Programlanabilir Kapı Dizisi
GAP	Genel Atama Problemleri
GPU	Grafik İşlemci Birimi
GPGPU	Genel Amaçlı Grafik İşlemci Birimleri
HAVELSAN	Hava Elektronik Sanayi A.Ş
MMAS	MAX-MIN Karınca Sistemi
NSGA	Baskın Olmayan Sıralama Genetik Algoritma
OpenCL	Açık Hesaplama Dili
PPSN 2014	International Conference on Parallel Problem Solving from Nature 2014
PSO	Parçacık Sürüsü Optimizasyonu
SAN-TEZ	Sanayi Tezleri Programı

SIMD	Tek İşlem, Çoklu Data
SPSO	Standart Parçacık Sürüsü Optimizasyonu
VEGA	Vektör Değerlendirmeli Genetik Algoritma

ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

1. GİRİŞ

Hava araçlarının keşif/gözetleme, arama/kurtarma görevlerini etkin bir şekilde gerçekleştirebilmeleri için, kısıtlı kaynakları verimli kullanma zorunlulukları bulunmaktadır. Bu zorunluluk, görevlerin öncesinde birden fazla amaç (görev riskinin en aza indirgenmesi, görev tamamlama süresinin en aza indirgenmesi, yakıt sarfiyatının en aza indirgenmesi, kaynakların dengeli kullanılması ve amortisman giderlerinin kontrol altında tutulması, vb.) gözetilerek optimizasyon temelli planlama yapılması ihtiyacını doğurmaktadır.

Görev Planlama işlemi en basit anlamda, bir küme/liste halinde verilen işlerin minimum zamanda (veya başka amaçlar gözetilerek) yerine getirilmesi için kaynak planlamasının yapılması olarak tanımlanabilir. Varsa öncelikli veya önemli işlerin ilk başta yerine getirilmesi, unsurlara iş ataması yapılırken her bir unsurun kabiliyetinin göz önünde bulundurulması vb. hususlar, problemin kısıtlarını (*constraint*) oluşturmaktadır. Dolayısıyla görev planlama problemi, aslen bir optimizasyon problemidir. Toplam işlem zamanının minimize edilmesinin yanı sıra, enerji tasarrufu, unsurların amortisman giderlerinin dengede tutulması vb. ek kaygıların da bulunması durumunda problem, Çok Amaçlı (*multi-objective*) Optimizasyon Problemi'ne dönüşmektedir.

Çok Amaçlı Optimizasyon Problemleri'nde birden fazla amaç önem sırasına göre çözüme etki ederken, çözüm kümesi mümkün olduğunca tüm amaçları karşılamaya yönelik oluşturulan çözümleri kapsamaktadır. Nihai amaç ise çözüm kümesi içerisinden son kullanıcının tercihi doğrultusunda en optimum çözümün seçilip uygulanmasıdır.

Bahsedilen ihtiyaçları karşılamak üzere ortaya çıkan Çok Amaçlı Optimizasyon Problemi bu tez kapsamında belirlenen en uygun Skalarizasyon yöntemi ile birden fazla Tek Amaçlı Atama Problemlerine indirgenerek her bir problem eş zamanlı olarak çözülmüştür. Atama Problemleri'nin çözümünde kullanılan yöntemler, üç ana başlık altında incelenebilir: "Deterministik Yöntemler", "Olasılık Temelli Yöntemler" ve "Buluşsal Yöntemler".

Buluşsal Yöntemler, bir problem türü üzerinde yoğun, dikkatli bir araştırma ve sağduyu ile o probleme özel, çoğunlukla en iyiye yakın veya pratik değeri olan çözüm bulabilen yaklaşımları ifade etmek için kullanılır. Buluşsal Yöntemler, deterministik yöntemlerin yeterli olmadığı durumlarda, atama problemlerinin özel yöntemlerle nasıl çözülebileceği üzerinde durulur. Bu yöntemler, paralelleştirme işlemine uygun olduğu için özellikle çok yüksek boyutlu problemlerin Çok Amaçlı Optimizasyon Problemi gibi ele alınabilmesine, çok çekirdekli işlemci mimarileri üzerinde aynı anda birçok alternatif çözümün bulunabilmesine olanak sağlamaktadır. Tez kapsamında, son yıllarda bilimsel hesaplama amacıyla da kullanılmaya başlanan ve grafik kartları üzerinde bulunan Genel Amaçlı Grafik İşlemci Birimleri (*General Purpose Graphics Processing Units - GPGPUs*) üzerinde, CUDA programlama altyapısı kullanılarak, paralelleştirme teknikleri uygulanarak, Genel Amaçlı Atama Problemleri, Çok Amaçlı Optimizasyon Problemi gibi ele alınarak ve buluşsal algoritmaların en başarılı örneklerinden Genetik Algoritma ile çözülmüştür.

Tez ile daha önceden de belirtildiği üzere atama problemleri üzerine sonuç çıkartımı yapılarak sonuca ulaşılmaya çalışılmıştır. Ele alınan yöntemler ile belirli girdi değerlerine karşı çıkış performansları maliyet, zaman, risk, vb. özellikler bakımından kıyaslanarak ele alınan kısıtlar dahilinde en iyi sonuca ulaşmaya yardımcı olacak yöntem belirlenerek T.C. Bilim Sanayi ve Teknoloji Bakanlığı tarafından desteklenen "Grafik İşlemci Birimleri Üzerinde Genel Atama Problemlerinin Çözümü" başlıklı ve 01568.STZ.2012-2 numaralı SAN-TEZ projesi kapsamında bir yazılım geliştirme arayüzü ortaya çıkarılmıştır.

Bu tez çalışmasından elde edilen optimizasyon algoritmaları ve yazılım geliştirme arayüzü, hava unsurları için mevcut milli görev planlama ürünlerinde kullanılarak, Milli Görev Planlama Sistemlerinin yeteneklerinin artırılması hedeflenmektedir. Tez ile elde edilen kazanımlar sadece hava unsurlarının görev planlaması için değil, milli savaş gemilerinde kullanılmak üzere geliştirilen yeni nesil Savaş Yönetim Sistemlerinde de kullanılabilecektir.

Bu amaçlar doğrultusunda ortaya atılan problem ve problemin çözümü için kullanılan buluşsal yöntemler ile Çok Amaçlı Optimizasyon Probleminin Tek Amaçlı Optimizasyon Problemi'ne indirgenmesinde kullanılan skalarizasyon yöntemleri 2. bölümde "Çok Amaçlı Optimizasyon Problemleri ve Buluşsal Yöntemler" başlığı altında incelenecektir.

3. bölümde, Paralel Programlama kavramı, grafik kartları üzerinde paralelleştirme yöntemi ve Grafik İşlemci Birimleri üzerinde Genel Amaçlı Hesaplamalar konusunda yapılan çalışmalar "Paralel Programlama ve GPGPUlar" başlığı altında ele alınacaktır.

Tez kapsamında uygulanan mimari yaklaşım ve elde edilen sonuçlar 4. bölümde gösterilecektir. Ayrıca, yapılan çalışma süresince karşılaşılan problemler ve çözümleri, kazanımlar ve implementasyon bu bölümde detaylı bir şekilde anlatılacaktır.

Son bölümde ise elde edilen bilgi birikimi ile sonuçlar tartışılacak ve gelecek dönemde yapılabilecek iyileştirmeler/geliştirmeler anlatılacaktır.

Ek olarak, tez kapsamında yapılan çalışmanın bir bölümü olan Çok Amaçlı Optimizasyon Problemleri'nde Skalarizasyon Tekniklerinin paralel implementasyonuna yönelik uygulama detaylarını anlatan "Parallel Cuda Implementation of the Desirability-Based Scalarization Approach for Multi-Objective Optimization Problems" başlıklı akademik metin "13th International Conference on Parallel Problem Solving from Nature (PPSN), Ljubljana 2014" konferansında yer alan "Student Workshop on Bioinspired Optimization Methods and their Application (BIOMA 2014)" çalıştayında sunulmuştur.

Diğer yandan, skalarizasyon tekniklerinin paralel implementasyonu ve buluşsal algoritmaların en bilinen kıyaslama problemleri üzerindeki performansını anlatan bir kitap bölümü çalışması ise "Parallel Implementation of Scalarization Approaches for Multi-objective Optimization Problems" başlığıyla "Advances in Evolutionary Algorithms Research" adlı kitapta yer almıştır.

Çok Amaçlı Optimizasyon Problemleri'nde skalarizasyon tekniklerinin paralel implementasyonu ve bir takım kıyaslama problemleri üzerinde performanslarının karşılaştırmalarını anlatan bir diğer çalışma ise Uluslararası Hesaplama ve Bilişim Dergisi olan "Informatica"'nın "Special Issue of Informatica on Bioinspired Optimization" adlı özel sayısında "Parallel Implementation of Desirability Function-based Scalarization Approach for Multi-objective Optimization Problems" başlığıyla yayınlanmıştır.

2. ÇOK AMAÇLI OPTİMİZASYON PROBLEMLERİ ve BULUŞSAL YÖNTEMLER

Çok Amaçlı Optimizasyon Problemleri, gündelik hayatta sıklıkla karşılaşılan; çözümleri de karmaşık yöntemlerin kullanımını gerektiren zor problemlerdir. Bu nedenle çok amaçlı optimizasyon, farklı disiplinlerdeki birçok araştırmacının üzerinde çalışmakta olduğu bir konu olup; bu tez çalışmasının da temelini teşkil etmektedir. Gündelik hayatta rastlanan ve kısıtları çok fazla olan bu problemlerin çözümünde, gradyant tabanlı olarak da bilinen klasik optimizasyon algoritmaları genellikle yetersiz (hatta uygulanamaz) kalmaktadır. Bu gibi problemlerin çözümünde, aslen sistematik deneme-yanılma esasına dayanan buluşsal (*heuristic*) yöntemlere başvurulmaktadır.

Buluşsal yöntemler:

- Klasik optimizasyon algoritmalarından farklı olarak, optimize edilecek fonksiyonun sürekli veya türevlenebilir olmasını gerektirmemeleri,
- Yine klasik optimizasyon algoritmalarından farklı olarak, optimize edilecek fonksiyon hakkında niteliksel bile olsa herhangi bir ön bilgi gerektirmemeleri,
- Kısıtları pratik, ancak başarılı bir şekilde ele almaları

nedeniyle, özellikle son 20 yıl içerisinde her alanda yaygın olarak kullanılmaya başlamıştır. Buluşsal yöntemlerin birçoğu:

- Doğadaki bir takım fenomenlerden (evrim teorisindeki doğal seleksiyon, cisimler arasındaki çekim kuvveti, dönen bir cisme etki ettiği düşünülen merkezkaç kuvveti, canlılardaki DNA'nın kopyalanması, vb.)
- Canlıların (kuş, balık ve böcek sürüleri; mikroorganizmalar, bitkiler, vb.) davranışlarından, veya
- Devinim içerisinde cansız yapıların (yağmur damlaları, akarsular, vb.) davranışlarından

ilham alınarak ve bu fenomenlerin/organizmaların/yapıların taklit edilmesi yoluyla geliştirilmiştir. Bu sınıftaki en yaygın yöntemler arasında Genetik Algoritma, Parçacık Sürüsü Optimizasyonu, Karınca Kolonisi Optimizasyonu, Farksal Evrim Algoritması örnek verilebilir.

Bu yöntemler, yukarıda belirtilmiş olan avantajlarının yanı sıra:

- Hem ayrık, hem sürekli optimizasyon problemlerine uygulanabilir türevlerinin olması,
- Çok modlu (*multimodal*), bir başka deyişle çok sayıda lokal optimum içeren karmaşık fonksiyonlar için de global optimuma ulaşabilmeleri,
- Aslen tek amaçlı optimizasyon problemleri için geliştirilmiş olmalarına karşın, çok amaçlı optimizasyon problemlerine yönelik versiyonlarının da bulunması,
- Çoğunlukla basit işlemler içermeleri dolayısıyla kolaylıkla programlanabilir donanım yapıları (örneğin FPGA) üzerinde gerçeklenebilir olmaları,
- Popülasyon tabanlı olmaları dolayısıyla da gerek yazılım, gerekse donanım olarak gerçeklenmelerinde paralelizasyona olanak sağlamaları

gibi nedenlerle de gerek zaman kısıtlaması olmayan, gerekse zaman kısıtlaması olan birçok uygulamada yaygın kullanım alanı bulmaktadırlar. Literatürde, oldukça karmaşık ve gereksinimleri hayli agresif olan problemlerde bile, buluşsal yöntemlerin başarılı olduğu bildirilmiştir. Doğadan ilham alan buluşsal yöntemlerin basitleştirilmiş planlama faaliyetleri için kullanımları mevcut olmakla beraber, bu yöntemlerin büyük ölçekli bir uygulamada kullanımı ve çok amaçlı yöntemlerin GPU implementasyonlarının başarımlarının karşılaştırılması, literatürde bulunmamaktadır.

Tez çalışması için ele alınan Çoklu Hava Unsuru Görev Planlama problemi, aşağıdaki tanımlar aracılığıyla betimlenebilir.

Tanımlar:

U_n (n={1,2,...,N}): n'inci Hava Unsuru

Q_n (n={1,2,...,N}): n'inci Hava Unsuru'nun (U_n'in) maksimum görev süresi (birimi: dakika)

J_n (n={1,2,...,N}): n'inci Hava Unsuru tarafından yapılacak olan iş (*task*)

F_n (n={1,2,...,N}): n'inci Hava Unsuru'nun birim görev maliyeti (birimi: $ / dakika)

P_n (n={1,2,...,N}): n'inci Hava Unsuru tarafından yapılacak olan işin (J_n'in) uzunluğu (birimi: dakika)

K_n (n={1,2,...,N}): n'inci Hava Unsuru tarafından yapılabilecek olan işin (J_n'in) tipi

R_m (m={1,2,...,M}): m'inci iş (*task*)

T_m (m={1,2,...,M}): m'inci işin (R_m'in) tipi

S_m (m={1,2,...,M}): m'inci işin (R_m'in) maliyeti (birimi: dakika)

V_m (m={1,2,...,M}): m'inci işin (R_m'in) tamamlanma süresi (birimi: dakika)

D_m (m={1,2,...,M}): m'inci iş (R_m) için maksimum tolere edilebilir tamamlanma süresi (birimi: dakika)

$Z_{n,m}$ (n={1,2,...,N}) ve (m={1,2,...,M}): m'inci işin (R_m'in), n'inci Hava Unsuru tarafından yapılması esnasındaki risk (birimi: birimsiz)

Çoklu Hava Unsuru görev planlama problemi, $\forall i$=1,2,...,N ve $\forall j$=1,2,...,M için

a. $\sum_{i=1}^{N} w(V_i)$ (taktik öneme göre ağırlıklandırılmış *makespan*)

b. $\max\{w(V_i) \mid i = 1,...,N\}$ (taktik öneme göre ağırlıklandırılmış *flowtime*)

c. $\Sigma_{i,j}\, Z_{i,j}$ (toplam görev riski)

d. $\sum_{i=1}^{N} F_i$ (toplam görev maliyeti)

değerlerinden, strateji olarak seçilen(ler)i minimize edecek,

Λ (toplam görev etkinliği)

değerini ise maksimize edecek şekilde (U_i, R_j) çiftlerinin oluşturulması (veya $i \rightarrow j$ atamalarının gerçekleştirilmesi) olarak tanımlanabilir. Yukarıdaki ifadelerde $w()$, işin taktik önem derecesine göre ağırlık değeri döndüren bir ağırlıklandırma fonksiyonudur.

Sözü edilen problem, literatürdeki genel atama probleminin (*general assignment problem* - *GAP*) bir türevidir. Ancak, Çoklu Hava Unsuru Görev Planlama probleminde:

- İşlerin ve Hava Unsurları'nın sayısının eşit olması gibi bir zorunluluk bulunmamaktadır.
- Kendisine hiç atama yapılmayan Hava Unsurları olabileceği gibi, bazı Hava Unsurları'na ardışık olarak gerçekleştirilmek birden fazla iş atanabilmektedir.
- Minimize edilecek değer, sadece süre olmayıp; yukarıda da görüldüğü üzere ağırlıklandırılmış *makespan, flowtime*, görev maliyeti veya görev riski değerlerinden en az biri olabilmektedir. Aynı zamanda, taktik önceliklere göre tanımlanacak olan görev etkinliği fonksiyonunun da maksimize edilmesi gerekmektedir. Birden fazla değerin aynı anda minimizasyonu/maksimizasyonu istenildiğinde problem, çok amaçlı bir optimizasyon problemine dönüşmektedir.

Problemin çözümü, bir takım varsayımlar ve kısıtlar altında gerçekleştirilmelidir. Aşağıda, söz konusu varsayımlar ve kısıtlar için temel birkaç örnek verilmiştir:

i) $\Sigma_j P_j < Q_n$ (Bir başka deyişle, U_n'ye atanmış olan işlerin toplam süresi, U_n'in maksimum görev süresini geçmemelidir)

ii) $V_m < D_m$ (Bir başka deyişle, R_m'nin tamamlanma süresi, maksimum tolere edilebilir tamamlanma süresini geçmemelidir)

iii) Belirli K_i'ler ve T_j'ler için (U_i, R_j) çiftinin oluşturulması (veya $i \rightarrow j$ ataması) yasaklı olabilir.

Tez kapsamında yukarıda tanımlamaları ve kısıtlamaları ile birlikte belirtilen problem en temel haliyle ele alınmıştır. Sel (2013), çalışmasında Deterministik Yöntemler, Olasılık Temelli Yöntemler ve Buluşsal Yöntemlerden olmak üzere bir çok algoritmayı Çoklu Hava Unsurları için Görev Planlama Probleminin çözümünde denemiş ve en iyi sonucu Genetik Algoritma ile elde ettiğini belirtmiştir. Bu sebeple tez çalışmasında problemin çözümünde diğer algoritmaların tekrar denenmesine gerek olmadan Genetik Algoritma kullanılmıştır. Ayrıca Çok Amaçlı Optimizasyon Problemindeki amaçlar bir potada eritilip tek amaca indirgenirken skalarizasyon tekniklerinden faydalanılmıştır. Sonraki bölümlerde tez kapsamında incelenen ve uygulanan skalarizasyon teknikleri anlatılacaktır.

2.1 Çok Amaçlı Optimizasyon Problemlerinde Skalarizasyon Teknikleri

Çok Amaçlı Optimizasyon Problemleri'nin çözümüne yönelik birçok yaklaşım vardır. En temel yaklaşım "Skalarizasyon", bir başka deyişle "bir araya getirme" yaklaşımıdır. Bu yaklaşım, tek bir amaç elde etmek için bütün amaçların bir araya getirilmesini esas alır (Marler ve Arora 2005). Böylece Çok Amaçlı Optimizasyon Problemleri, Tek Amaçlı Optimizasyon Problemi'ne indirgenerek çözülür. En yaygın bilinen skalarizasyon tekniği "Ağırlıklı Toplam" tekniğidir (Koski ve Silvennoinen 1987, Saramago ve Steffen Jr 1998, Marler ve Arora 2010).

Skalarizasyon teknikleri, NSGA (Non-Dominated Sorting Genetic Algorithm) (Srinivas ve Deb 1995), NSGA-II (Deb vd. 2002) ve VEGA (Vector Evaluated Genetic Algorithm) (Schaffer 1985) vb. gibi oldukça güçlü Çok Amaçlı Optimizasyon Algoritmaların gelişiminden önce, 1980'ler ve 1990'ların başlarında popüler oldu. Bu algoritmalar ortaya çıktıktan sonra skalarizasyon tekniklerinden artık eskisi kadar etkili olmadığı düşünülerek vazgeçildi. Fakat daha sonra, özellikle 2000'lerde çok çekirdekli mimarilerin ortaya çıkmasıyla bilim adamları doğru implement edildiklerinde skalarizasyon tekniklerinin paralelleştirme için her zaman faydalı ve kullanılabilir olduklarını düşünerek yeniden skalarizasyon teknikleri üzerinde çalışmaya başladılar. Böylece skalarizasyon teknikleri, Çok Amaçlı Optimizasyon Problemleri'nin çözümünde paralelleştirme yöntemleri ile birlikte kullanılmaya başlandı.

Tez kapsamında yapılan çalışmada ise, öncelikle Ağırlıklı Toplam tekniği denenmiş ancak, sonraki bölümlerde bahsedilen sebeplerden dolayı bu teknik yetersiz kalmıştır ve yerine "Çekicilik Fonksiyonu" tekniği kullanılmıştır.

2.1.1 Ağırlıklı toplam tekniği

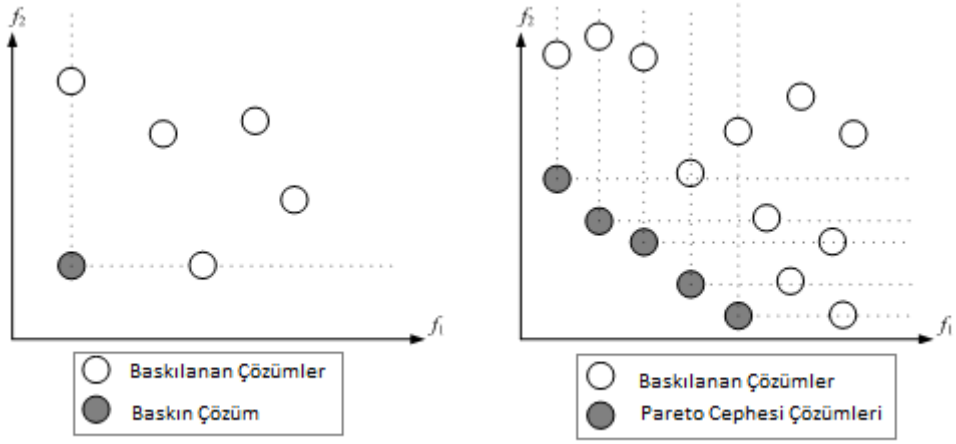
Çok Amaçlı Optimizasyon Problemi'nin çözümünde "tercihe dayalı" sonuçlar ortaya çıkmaktadır (Trautmann ve Mehnen 2009). Öte yandan skalarizasyon tekniği kullanılarak tek amaca indirgenen problemin çözümünden tek bir sonuç ortaya çıkar. Skalarizasyon tekniği kullanılarak tek amaca indirgenen problem, Çok Amaçlı Optimizasyon Problemi'nin sadece bir örneğidir. Diğer bir örneği elde edebilmek için, skalarizasyon sürecinde kullanılan parametreler değiştirilir ve ortaya çıkan bir öncekinden farklı bir örnek çözülür.

Örnek olarak, iki amaçlı optimizasyon problemi ele alındığında ve bu probleme Ağırlıklı Toplam tekniği uygulandığında ortaya çıkan amaç fonksiyonu Eşitlik (2.1)'teki gibi olmaktadır.

$$c = w_1 f_1 + (1 - w_1) f_2 \tag{2.1}$$

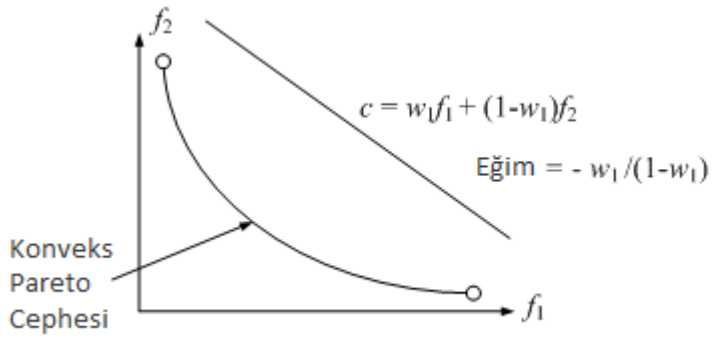
Bu eşitlikteki $w1$ ağırlık değeridir ve 0 ile 1 arasında tanımlı gerçek sayıdır. Bu ağırlık değeri değiştirilerek yeni bir tek amaçlı optimizasyon problemi elde edilir.

Ağırlık değeri değiştirilerek çözülen Tek Amaçlı Optimizasyon Problemleri'nden elde edilen sonuçlar arasından bir sonucun uygunluğuna, o sonucun Pareto uzayı içerisindeki baskınlığına bakılarak karar verilir. Şekil 2.1'de iki amaçlı optimizasyon probleminin çözümüne ait sonuç kümesi, baskın sonuç ve sonuçlar (Pareto Cephesi) ile baskılanan sonuçlar gösterilmiştir.

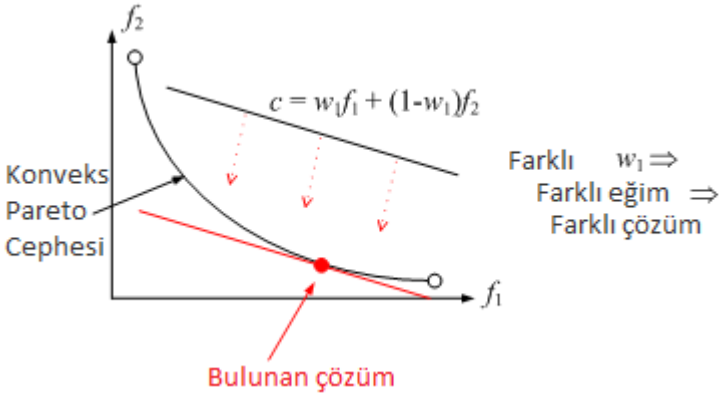


Şekil 2.1 Baskınlık ve Pareto Cephesi resimsel gösterimi

Çok Amaçlı Optimizasyon Problemleri'nde amaç, tercihe dayalı bir sonuç kümesi elde etmektir. Örnek olarak ele alınan iki amaçlı optimizasyon probleminin çözülmesiyle elde edilen sonuç kümesi içerisinde bulunan en uygun sonuca "Baskın Sonuç (Dominating Solution)", diğer sonuçlara da "Baskılanan Sonuçlar (Dominated Solutions)" adı verilir (Şekil 2.1). Birden fazla baskın sonucun varlığı ile "Pareto Cephesi (Pareto Front)" oluşur (Şekil 2.1). Ağırlıklı Toplam tekniği kullanılarak çözüm kümesi elde edebilmek için eşitlik(2.1) kullanılır ve ağırlık değeri 0 ile 1 arasında sistematik bir şekilde değiştirilir. Bu yöntem ile her bir ağırlık (w_1) değeri için yeni bir sonuç bulunmuş olur.

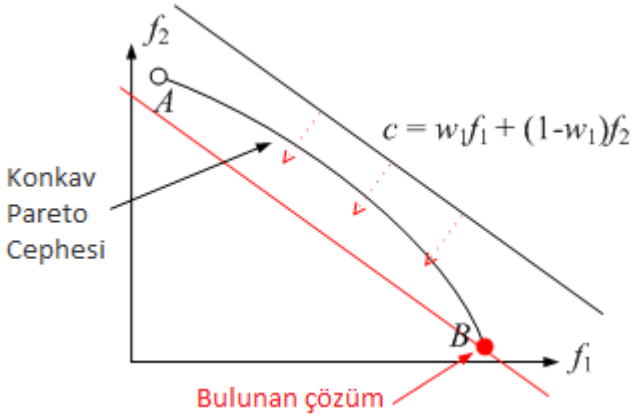


Şekil 2.2 Konveks Pareto Cephesi için Ağırlıklı Toplam tekniği kullanılarak sonucun bulunması

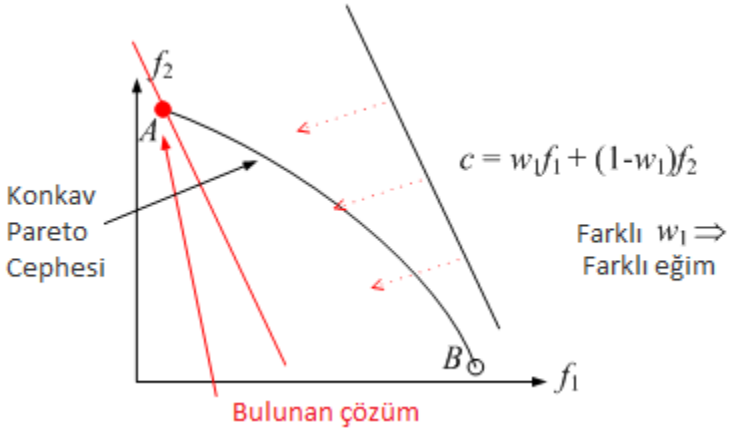


Şekil 2.3 Konveks Pareto Cephesi için Ağırlıklı Toplam tekniği kullanılarak ağırlık değerinin değiştirilmesiyle diğer sonuçların bulunması

Şekil 2.2'de gösterildiği üzere eşitlik(2.1) Pareto uzayında (f_1f_2 düzleminde) bir doğruyu belirtir. Optimizasyon işlemi boyunca $c = w_1f_1 + (1 - w_1)f_2$ eşitliği minimize edilerek, eşitliğin belirttiği doğru sabit eğimle orijine doğru kaydırılır. Doğru ile Pareto Cephesi eğrisinin kesişimi sonucu vermektedir. Ağırlık değeri doğrunun eğimini belirtir ve ağırlık değerinin değiştirilmesi ile farklı eğimde bir doğru elde edilir. Şekil 2.3'de ağırlık değeri değiştirilmiş, dolayısıyla farklı bir eğimde bir doğru ile Konveks Pareto Cephesi için bulunan diğer bir sonuç gösterilmiştir.



Şekil 2.4 Konkav Pareto Cephesi için Ağırlıklı Toplam tekniği kullanılarak sonucun bulunması



Şekil 2.5 Konkav Pareto Cephesi için Ağırlıklı Toplam tekniği kullanılarak ağırlık değerinin değiştirilmesiyle diğer sonucun bulunması

Pareto Cephesinin konveks olduğu durum için Ağırlıklı Toplam tekniği yukarıda bahsedildiği gibi uygulanır ve sonuçlar elde edilir. Bir diğer yandan, Pareto Cephesinin konkav olma ihtimali de söz konusudur ve böyle bir durum için $c = w_1 f_1 + (1 - w_1) f_2$ eşitliğinin belirttiği doğru sabit eğimle minimize edildiğinde bulunan sonuç şekil 2.4'te gösterildiği gibi B noktasıdır. Ağırlık değeri değiştirilerek elde edilen diğer bir sonuç ise A noktasıdır. Böyle bir durumda, ağırlık değeri ne kadar değiştirilse de A ve B noktalarından farklı bir sonuç elde edilemeyecektir (Burachik vd. 2014). Şekil 2.4 - 2.5 bu durumu resimsel olarak ifade etmektedir. Gerçek problemlerde Pareto Cephesinin konveks veya konkav formda olacağını önceden bilmek mümkün değildir. Bu yüzden Ağırlıklı Toplam tekniği gerçek problemlere tam olarak uygulanamamaktadır.

2.1.2 Çekicilik fonksiyonu tekniği

Tez kapsamında yapılan çalışmada öncelikle Ağırlıklı Toplam tekniği uygulanmış fakat bir önceki bölümde anlatıldığı gibi başarısız olduğu görülmüştür. Bu yüzden bir başka skalarizasyon tekniği olan Çekicilik Fonksiyonu tekniği uygulanmıştır (Altınöz vd. 2013). Bu teknik ilk olarak 1965 yılında Harrington tarafından ortaya atılmıştır. Çekicilik Fonksiyonu tekniğinin önerilmesinden sonra, Derringer ve Suich (1980) bu teknik için temel eşitlikler olan değişik formülasyonlar sundular. Tek taraflı ve çift taraflı Çekicilik Fonksiyonu olarak adlandırılan formülasyonlardan: Eşitlik(2.2) ve Eşitlik(2.3)'te gösterilen formülasyon tek taraflı Çekicilik Fonksiyonu ve Eşitlik(2.4)'te

gösterilen formülasyon ise çift taraflı Çekicilik Fonksiyonu tanımlamak için kullanılmaktadır.

Eşitliklerde kullanılan parametreler aşağıdaki gibidir:

- y : Giriş parametresi (Amaç fonksiyonu değeri)
- $h_{\min}$: Çekicilik Fonksiyonu için minimum amaç fonksiyonu değeri
- $h_{\max}$: Çekicilik Fonksiyonu için maksimum amaç fonksiyonu değeri
- h_{med} : Çift taraflı Çekicilik Fonksiyonu için ortalama amaç fonksiyonu değeri

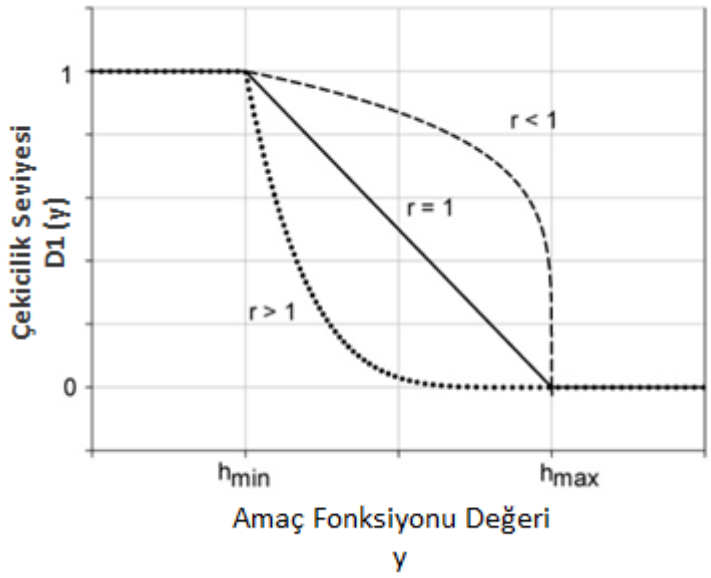
$$d_1(y) = \begin{cases} 1 & y < h_{\min} \\ \left(\dfrac{y - h_{\max}}{h_{\min} - h_{\max}} \right)^r & h_{\min} < y < h_{\max} \\ 0 & y > h_{\max} \end{cases} \tag{2.2}$$

$$d_2(y) = \begin{cases} 0 & y < h_{\min} \\ \left(\dfrac{y - h_{\min}}{h_{\max} - h_{\min}} \right)^r & h_{\min} < y < h_{\max} \\ 1 & y > h_{\max} \end{cases} \tag{2.3}$$

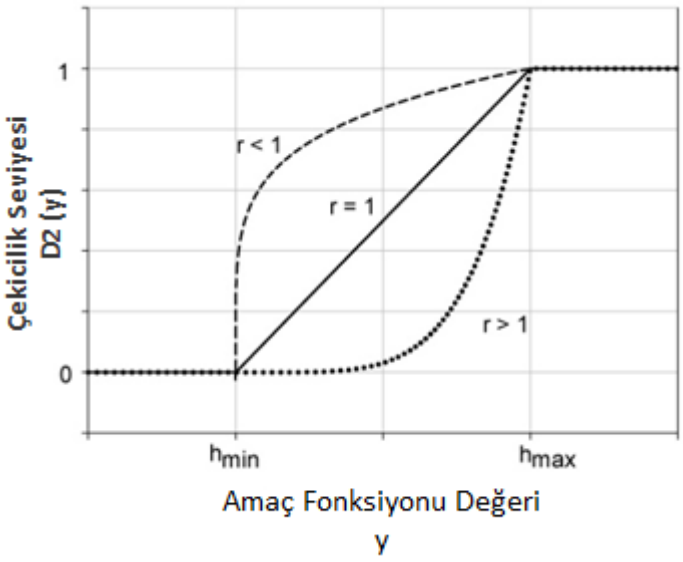
$$d_3(y) = \begin{cases} 0 & y < h_{\min} \\ \left(\dfrac{y - h_{mi}n}{h_{med} - h_{\min}} \right)^t & h_{\min} < y < h_{med} \\ \left(\dfrac{y - h_{\max}}{h_{med} - h_{\max}} \right)^s & h_{med} < y < h_{\max} \\ 0 & y > h_{\max} \end{cases} \tag{2.4}$$

Bir amaca ait bir değerin ne derece çekici olduğunu belirlemek için "Çekicilik Seviyesi" kavramı kullanılır. Çekicilik seviyesi d(y) =1 o değerin bütünüyle çekici, d(y)=0 ise bütünüyle çekici olmayan durumunu belirtir. Bu durumda d_1, minimizasyon problemleri

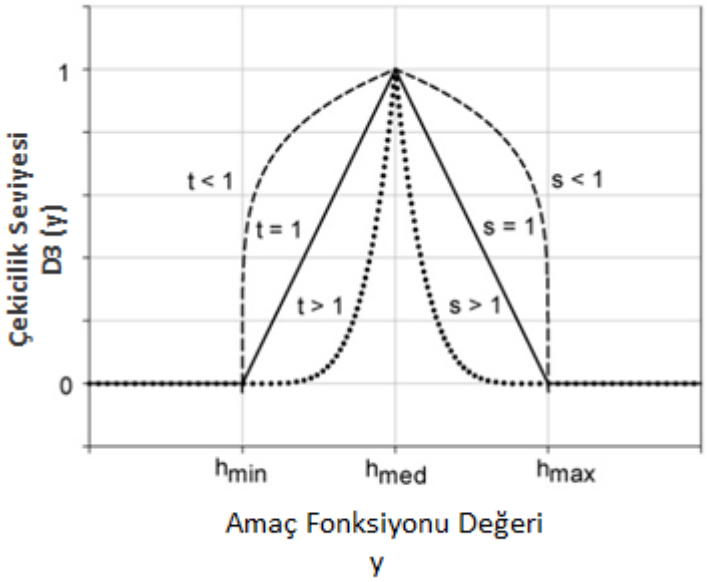
için kullanılan tek taraflı Çekicilik Fonksiyonunu ifade eder. Fonksiyonun eğimi *r*, *t* ve *s* parametreleri ile belirlenir. Bir Çekicilik Fonksiyonu bu parametreler ile sadece bir doğru ile değil bir yay ile de tanımlanabilir (Altınöz vd. 2013). Şekil 2.6 - 2.8'de, tek taraflı ve çift taraflı Çekicilik Fonksiyonları ile bu fonksiyonların doğru ve yay ile tanımlamaları gösterilmiştir.



Şekil 2.6 Eşitlik (2.2)'te ifade edilen Çekicilik Fonksiyonunun resimsel gösterimi



Şekil 2.7 Eşitlik (2.3)'te ifade edilen Çekicilik Fonksiyonunun resimsel gösterimi

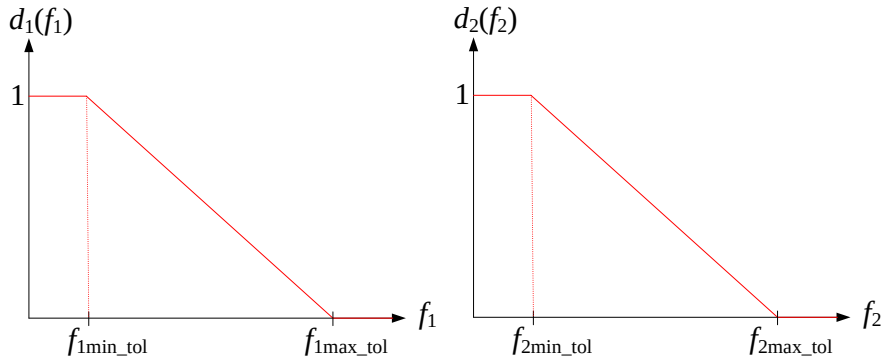


Şekil 2.8 Eşitlik (2.4)'te ifade edilen Çekicilik Fonksiyonunun resimsel gösterimi

Çekicilik Fonksiyonunun ana fikri aşağıda belirtildiği gibidir:

- Çekicilik Fonksiyonu, gerçek amaç fonksiyonu değerleri ile [0,1] arasında değişen değerler arasında bir eşleştirmedir.
- Bir Çekicilik Fonksiyonu, Çok Amaçlı Optimizasyon Problemlerinde sadece bir amaca karşılık gelir ve ilgili amaç fonksiyonunun değerlerini [0,1] aralığına dönüştürür.
- Her bir amaç fonksiyonunun minimizasyonuna yönelik isteğe bağlı olarak (örneğin: minimum/maksimum değerler) ilgili Çekicilik Fonksiyonu oluşturulur.
- Nihai çekicilik değeri tüm Çekicilik Fonksiyonlarının geometrik ortası olarak tanımlanır ve bu değer maksimize edilir.

İki amaçlı optimizasyon problemi için (f_1 ve f_2 fonksiyonları minimize edilen) oluşturulan Çekicilik Fonksiyonları $d_1(f_1)$ ve $d_2(f_2)$ şekil 2.9'da gösterildiği gibidir.

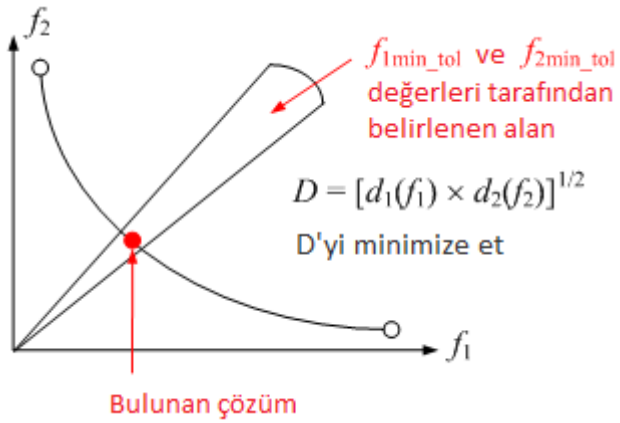


Şekil 2.9 İki Amaçlı Optimizasyon Problemleri için kullanılan Lineer Çekicilik Fonksiyonu resimsel gösterimi

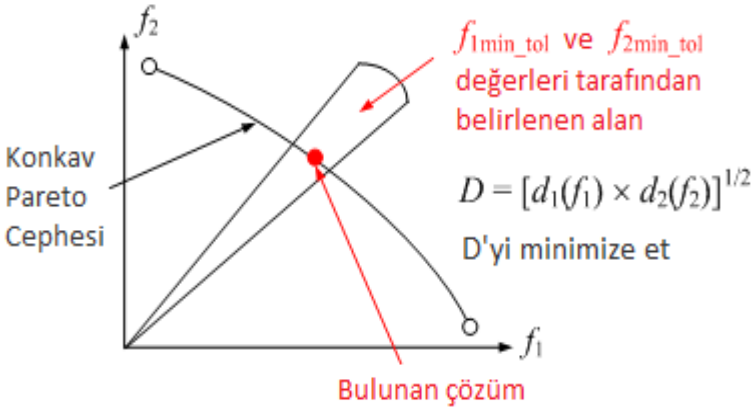
Çekicilik Fonksiyonunun oluşturduğu şeklin sistematik bir biçimde değiştirilmesi ile Pareto Cephesi elde edilir (Altınöz vd. 2013). Bunun için de aşağıdaki adımlar uygulanır:

- Şekil 2.9'da gösterilen f_{1max_tol} ve f_{2max_tol} parametreleri sonsuza çekilir.
- f_{1min_tol} ve f_{2min_tol} parametreleri sistematik bir biçimde değiştirilir.

Bahsedilen bu yöntem kullanılarak, konveks veya konkav formda olmasına bakılmaksızın Pareto Cephesi bulunur. İki Amaçlı Optimizasyon Problemi için sunulan bu yöntem şekil 2.10'da gösterildiği gibidir. f_{1min_tol} ve f_{2min_tol} parametreleri ile sektör belirlenir ve bu sektör içerisinden sonuç bulunur. Bulunan sonuç çekicilik değerlerinin geometrik ortalaması alındıktan sonra elde edilen noktaya karşılık gelmektedir. Şekil 2.11'den de görülebileceği üzere, Çekicilik Fonksiyonu tekniği ile Pareto Cephesi konkav formda bile olsa sonuç bulunabilmektedir. Böylelikle Çekicilik Fonksiyonu tekniği, Pareto Cephesinin hangi formda olacağı önceden bilinemeyen gerçek problemlerde bile rahatlıkla uygulanabilmektedir. Bir başka deyişle, Ağırlıklı Toplam tekniğinde olduğu gibi Çekicilik Fonksiyonu tekniği konkav formdaki Pareto Cephelerine uygulandığında başarısız sonuç vermemektedir (Altınöz vd. 2013).



Şekil 2.10 Konveks Pareto Cephesi için Çekicilik Fonksiyonu tekniği kullanılarak elde edilen sonuç

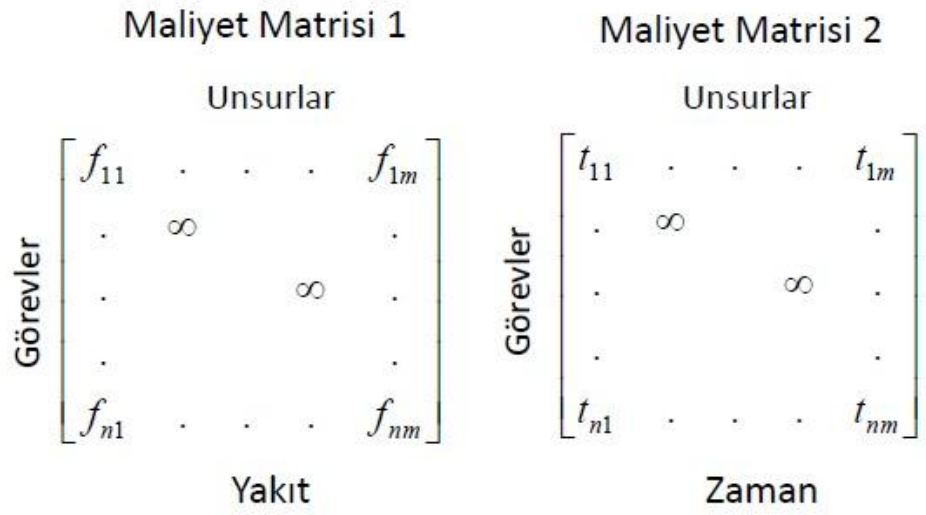


Şekil 2.11 Konkav Pareto Cephesi için Çekicilik Fonksiyonu tekniği kullanılarak elde edilen sonuç

Örnek olarak ele alınan görev planlama probleminde Çekicilik Fonksiyonu Temelli Skalarizasyon Tekniği aşağıdaki gibi uygulanmaktadır.

Skalarizasyon süreci, daha öncede bahsedildiği gibi problemde belirlenen amaç sayısı kadar maliyet matrisleri oluşturularak başlayacak, daha sonra oluşturulan bu matrislerden Çekicilik Fonksiyonu tekniğine ait skalarizasyon süreci işletilecek ve bu süreç sonunda bir tane nihai çekicilik matrisi elde edilecektir. Örnek görev planlama probleminde, harcanan yakıtın minimize edilmesi ve görev tamamlama süresinin minimize edilmesi olmak üzere iki farklı amaç bulunmaktadır. Bu iki amaca ait maliyet matrisleri şekil 2.12'deki gibidir.

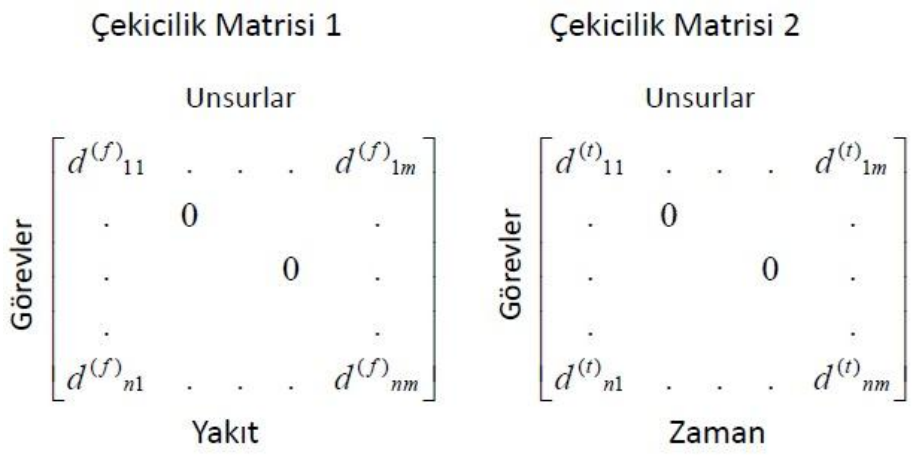
Her bir maliyet matrisi oluşturulurken, matrisin ∞ olmayan elemanları için i. görevin j. unsur tarafından gerçekleştirilmesinin reel maliyeti hesaplanarak ilgili yere yazılmaktadır. Matristeki ∞ olarak yazılan eleman, ilgili hava unsuru için icra edilebilir bir görevin planlanamadığını göstermektedir; bu duruma verilebilecek en iyi örnek unsurun herhangi bir rota üzerinden uçuş yapmadan bulunduğu meydanda kalması olarak verilebilir.



Şekil 2.12 İki amaçlı optimizasyon problemi için oluşturulan maliyet matrisleri

Maliyet matrisleri tanımlandıktan sonra, her bir maliyet kriteri için tanımlanmış olan şekil 2.9'da gösterildiği gibi "Çekicilik Fonksiyonları" kullanılarak çekicilik değerleri hesaplanmaktadır, böylece her bir maliyet matrisi için bir çekicilik matrisi hesaplanmış olur.

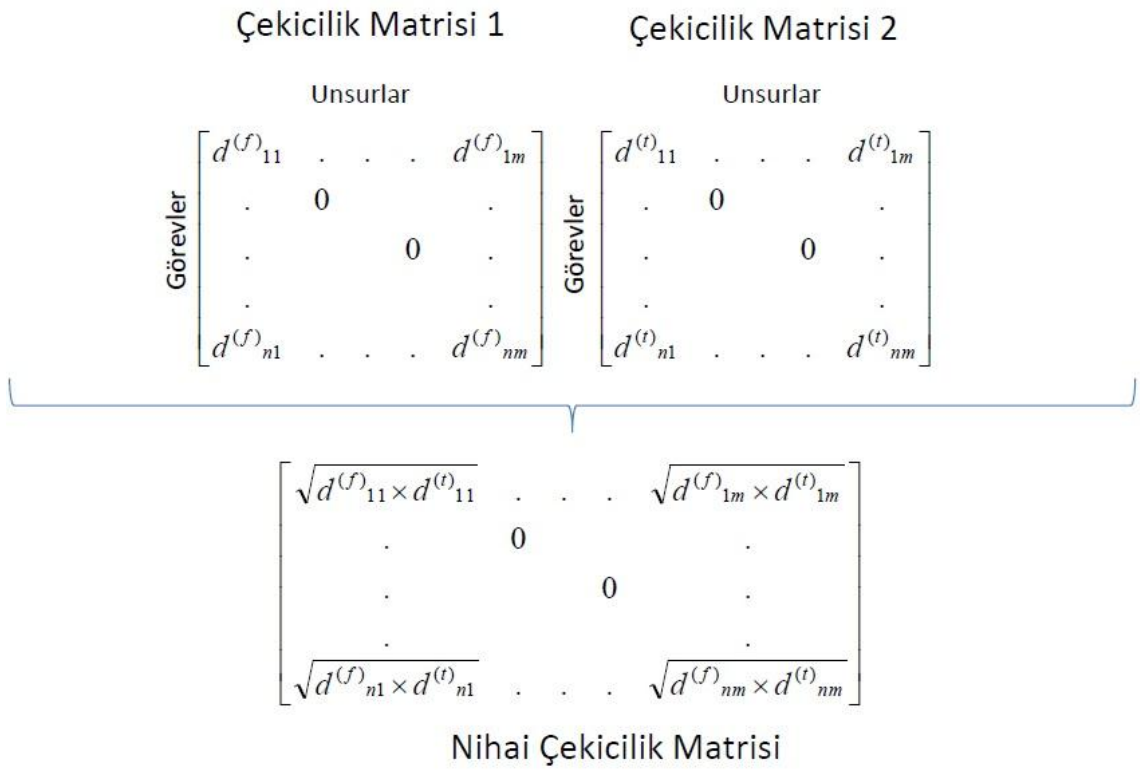
Çekicilik Fonksiyonları'nın tanımı gereği maliyet matrisindeki her bir eleman için hesaplanan çekicilik değeri 0'dan küçük ve 1'den büyük olamaz ve reel maliyeti sonsuz olan elemanın çekicilik değeri 0 olmalıdır. Yani, maliyet matrisinde bulunan bir görev unsur eşleşmesinin o amaç için hesaplanan reel maliyet değeri istenen amaca ne kadar uygunsa, o eşleşmenin çekicilik matrisindeki değeri de 1'e o kadar yakın olmaktadır. "Çekicilik Fonksiyonları" kullanarak hesaplanan çekicilik matrisleri şekil 2.13'de gösterilmiştir.



Şekil 2.13 İki amaçlı optimizasyon problemi için oluşturulan çekicilik matrisleri

Her bir amaç için reel maliyet matrislerinden elde edilen çekicilik matrislerinden son olarak bir tane nihai çekicilik matrisi hesaplanır. Bu hesaplama ise daha önce anlatıldığı gibi geometrik orta kullanılarak her bir eleman için yapılmaktadır. Örnek nihai çekicilik matrisi şekil 2.14'deki gibidir.

Skalarizasyon süreci sonunda elde edilen nihai çekicilik matrisindeki her bir elemanı (nihai çekicilik değeri) maksimize edecek şekilde optimizasyon süreci başlatılır ve bu süreç sonunda hesaplanan en uygun görevler unsurlara atanarak problem çözülmüş olur. Çekicilik Fonksiyonu Temelli Skalarizasyon Tekniği'nin tez kapsamında uygulandığı adım ile optimizasyon sürecine ait detaylar dördüncü bölümde anlatılmıştır.



Şekil 2.14 Çekicilik matrislerinden elde edilen nihai çekicilik matrisi

3. PARALEL PROGRAMLAMA VE GPGPULAR

3.1 Paralel Programlama

Parçalara bölünmüş ve uyarlanmış aynı görevin çoklu işlemcilerde eş zamanlı olarak işletilmesine "Paralel Hesaplama/ Programlama" adı verilir. Paralel Programlamadaki temel amaç sonuçları daha hızlı elde ederek daha kısa zamanda işin bitirilmesi veya aynı zamanda daha fazla iş yapılmasıdır. Paralellik veya paralelleştirme birkaç farklı şekilde yapılabilir.

Çizelge 3.1 Paralellik türleri (Akçay vd. 2011)

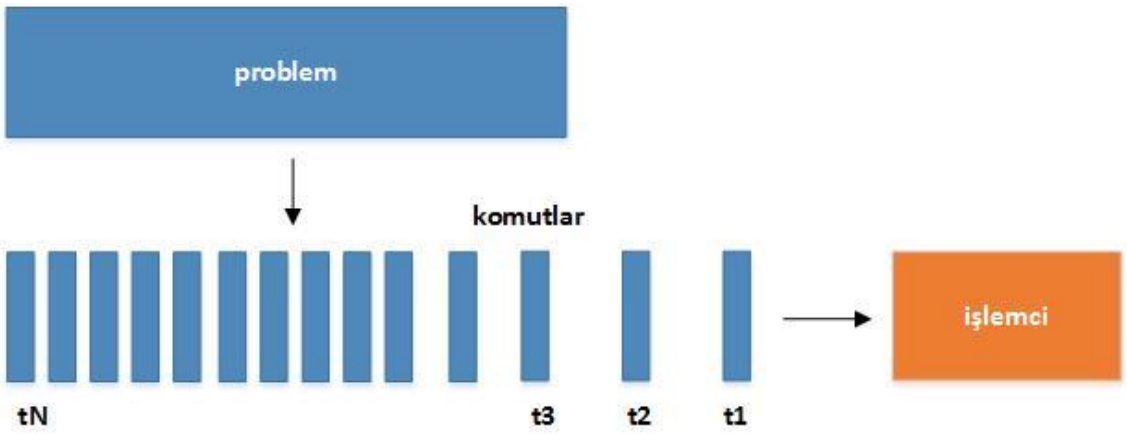
Paralelleştirme Türü	Gerçekleştirildiği Platform
Bit Düzeyinde Paralellik (Bit - Level Parallelism)	İşlemci İçinde
Komut Düzeyinde Paralellik (Instruction- Level Parallelism)	İşlemci İçinde
Veri Düzeyinde Paralellik (Data Parallelism)	İşlemci Dışında (CPU)
Görev Paylaşımlı Paralellik (Task Parallelism)	İşlemci Dışında (CPU, GPU)

Çizelge 3.1'de görüldüğü üzere paralellik 4 sınıfa ayrılır (Akçay vd. 2011). Bit Düzeyinde Paralellik ve Komut Düzeyinde Paralellik türleri işlemci içindeki işlenen bit miktarını ve çalıştırılan komut sayısını belirtirken; Veri Düzeyinde Paralellik ve Görev Paylaşımlı Paralellik türleri işlemci dışında sırasıyla aynı görevin farklı veri üzerinde çalıştırılması ve aynı/farklı veri üzerinde farklı görevlerin çalıştırılmasını belirtir.

Paralel Programlamayı Seri Programlamaya göre daha performanslı kılan en büyük etken Paralel Programlamanın çalışma şeklidir. Kullanılan programlama dili veya donanım ne olursa olsun Paralel Programlamanın çalışma şekli değişmez.

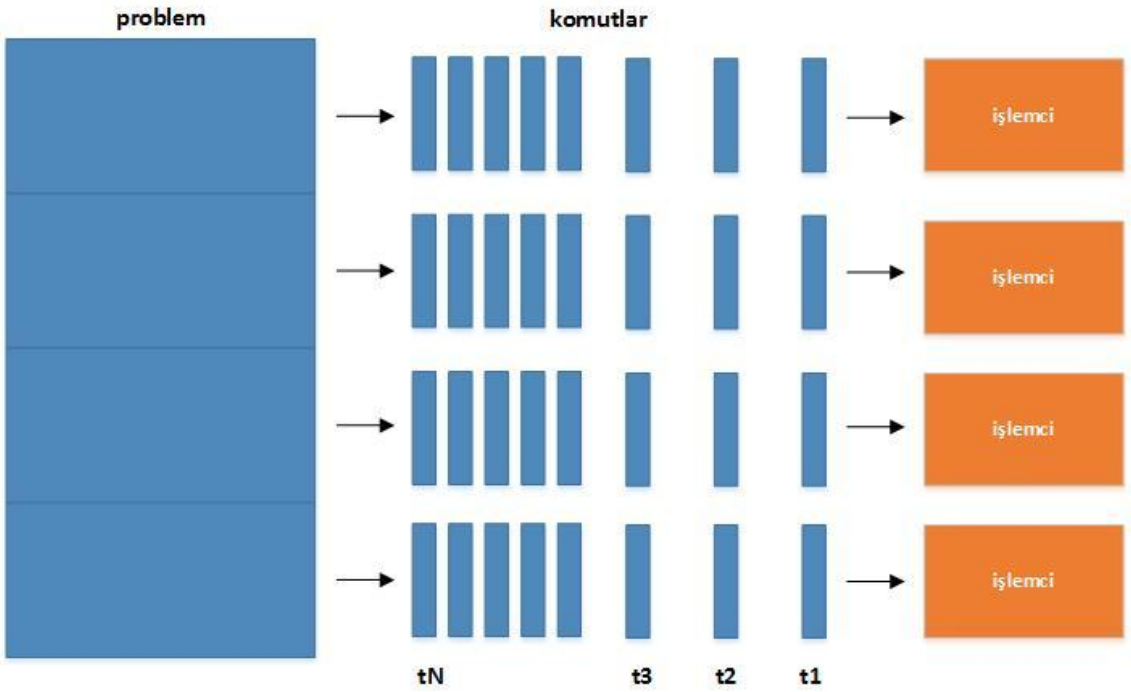
Şekil 3.1 - 3.2'de (Altıntaş ve Yegenoğlu 2011), Seri Programlama ve Paralel Programlama çalışma şekilleri gösterilmiştir. Seri Programlamanın çalışma prensibi temelde birbiri ardına gerçekleşen işlemler üzerine dayalıdır. Bir başka anlatımla, Seri Programlamada bir işlem bitmeden diğer bir işlem başlatılamaz. Öte yandan Paralel Programlama da böyle bir durum söz konusu olamaz. Çünkü Paralel Programlama da her bir işlem ayrı bir çekirdekte yapıldığından bir işlemin başlaması için diğer bir işlemin bitmesi beklenmez.

Şekil 3.1'de gösterilen Seri Programlama çalışma prensibinde, parçalara ayrılmış olan problemde yapılacak olan her bir işlem için T kadar zaman gerektiği düşünülürse, N tane işlem için kabaca T * N kadar zaman gerekir. Seri Programlama da yapılacak olan işlem sayısı ile toplam geçen süre arasında doğru orantı vardır ve bu da Seri Programlamanın yüksek performans gerektiren uygulamalarda kullanımına engel teşkil etmektedir.



Şekil 3.1 Seri programlamanın çalışma prensibi (Altıntaş ve Yegenoğlu 2011)

Diğer yandan Paralel Programlamada problem parçalara ayrıldıktan sonra, her bir parça ayrı bir çekirdek üzerinde işlenir. Çekirdekler aynı görevi farklı problem parçacıkları için gerçekleştirirken, işlem süresinde herhangi bir artış gözlenmez. Bir diğer deyişle Veri Düzeyinde Paralellik gerçekleştirilmiş olur. Paralel Programlamanın çalışma prensibi şekil 3.2'de gösterildiği gibidir.

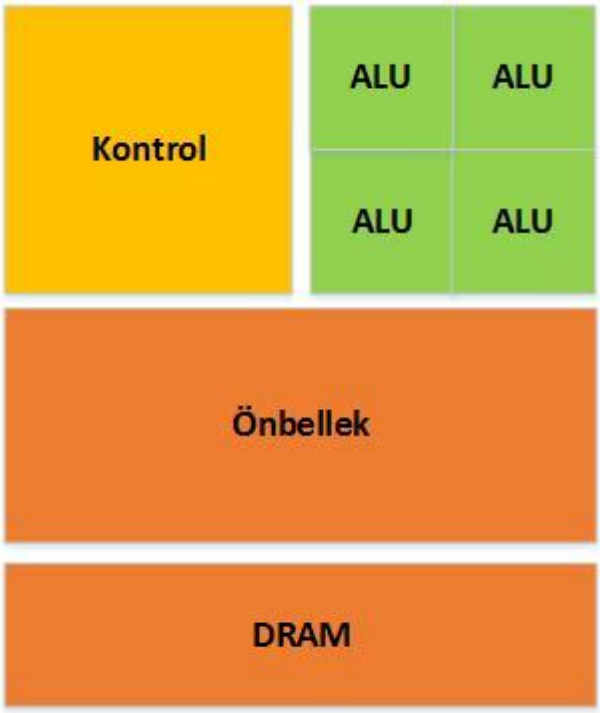


Şekil 3.2 Paralel programlamanın çalışma prensibi (Altıntaş ve Yegenoğlu 2011)

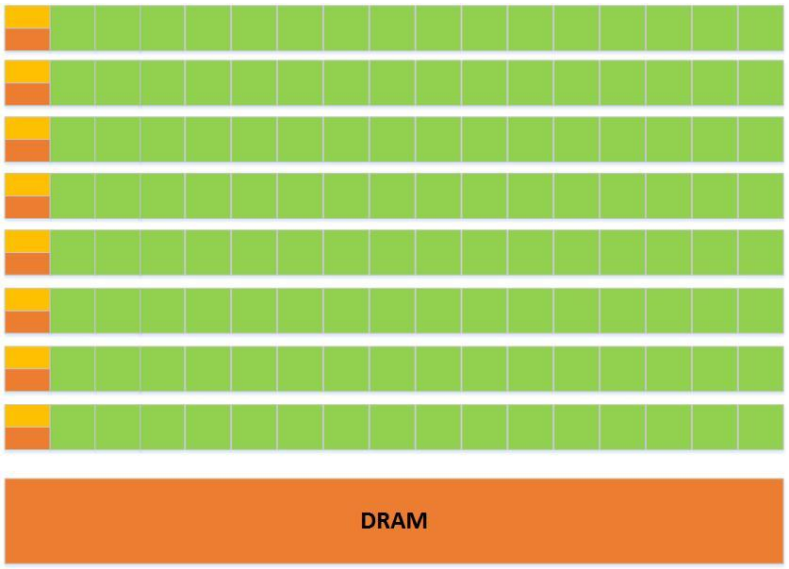
Bu tez kapsamında Veri Düzeyinde Paralellik türü üzerinde çalışılmış olup, Grafik İşlemci Birimleri'nden faydalanılmıştır.

3.1.1 Grafik İşlemci Birimi (GPU)

Grafik İşlemci Birimi (*Graphic Processing Unit* - GPU), bilgisayarlarda grafik oluşturma için kullanılan bir aygıttır. CPU ile kıyaslandığında içinde çok daha fazla sayıda transistör barındıran GPU, üzerinde bulunan çok sayıda işlemci sayesinde matris işlemlerini CPU dan çok daha hızlı yapabilmektedir.



Şekil 3.3 CPU mimarisi (Anonim 2009)



Şekil 3.4 GPU mimarisi (Anonim 2009)

Şekil 3.3 - 3.4 (Anonim 2009)'te görüldüğü gibi CPU ve GPU mimarisi birbirinden oldukça farklıdır. Aşağıda CPU ile GPU arasındaki temel farklar belirtilmiştir:

- Bilgisayarın temel işlemci birimi CPU iken, GPU sadece CPU'nun yükünü hafifletmek için kullanılır.

- GPU'nun esas ortaya çıkış amacı CPU'nun yerini almak değil, grafik işlemlerinin daha hızlı yapılabilmesini sağlamaktır.
- CPU'da en fazla birkaç çekirdek bulunabilirken, GPU'da bu sayı on binler mertebesine çıkabilir.
- GPU yüksek çekirdek sayısı sayesinde bölünebilir durumdaki büyük ölçekli verileri CPU'ya oranla çok daha kısa sürede işleyebilir.

3.1.2 CUDA

CUDA, GPU'lar için NVIDIA firması tarafından geliştirilen bir yazılım geliştirme platformudur. C programlama dili üzerinde eklenti olarak kullanıma sunulmuştur ve ilk CUDA Geliştirici Seti 2007 yılında piyasaya sürülmüştür. CUDA sözdizimi kuralları C programlama diline oldukça yakındır.

Şekil 3.5 ve şekil 3.6 (Buck ...?)'da gösterildiği üzere C ile CUDA dilleri bir örnek üzerinden karşılaştırılmıştır. Örneğe göre bir vektörün her bir elemanına sabit bir sayı eklenmektedir. Aynı programın C implementasyonunda seri programlama mantığının işleyişi, CUDA implementasyonunda ise paralel programlama mantığının işleyişi gösterilmektedir.

```
void increment_cpu(float *a, float b, int N)
{
    for(int idx =0; idx<N; idx++)
        a[idx] = a[idx]+b;
}
```

```
_global_ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    if(idx < N)
        a[idx] = a[idx] + b;
}
```

Şekil 3.5 C ile CUDA dilinin karşılaştırılması (Buck ...?)

Şekil 3.5'te görüldüğü üzere, örnekteki vektörün her elemanına sabit bir sayı eklemek için, programın C implementasyonunda vektörün tüm elemanları üzerinden bir döngü ile geçilmesi gerekirken, CUDA implementasyonunda böyle bir işlem söz konusu değildir. Çünkü vektörün her bir elemanına sabit bir sayı eklenmesi işini GPU üzerindeki ayrı çekirdekler yapmaktadır. Böylece programın C implementasyonunda bir

biri ardına gerçekleşen işlemler, CUDA implementasyonunda eş zamanlı olarak gerçekleştirilebilmektedir. CUDA diline ait çeşitli yerleşik değişkenler ve notasyonlar yardımı ile bu işlem için kaç çekirdek kullanılacağı, çekirdeklerin nasıl bir mimari ile çalıştırılacağı ve her bir çekirdeğe ait görevin ne olacağı belirtilir.

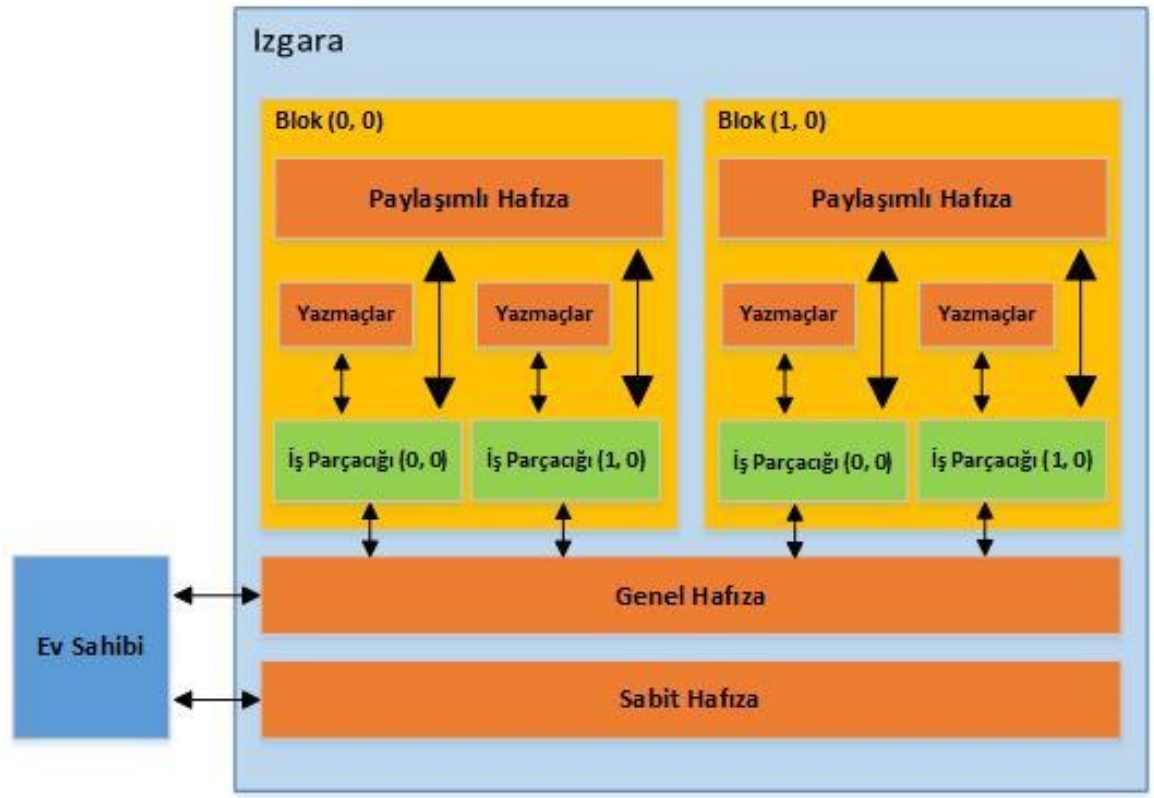
```
void main()
{
    .....
        increment_cpu(a,b,N);
}
```

```
void main()
{
    ....
    dim3 dimBlock(blocksize);
    dim3 dimGrid(gridsize);
    increment_gpu<<<dimGrid,dimBlock>>>(a,b,N);
}
```

Şekil 3.6 C ile CUDA dilinin karşılaştırılması (Buck ...?)

Şekil 3.6'da daha önce bahsedilen fonksiyonların çağırılma biçimleri gösterilmiştir. Görüldüğü gibi CUDA implementasyonundaki fonksiyon çağırımı ile C implemantasyonundaki fonksiyon çağırımı arasında belirgin bir fark vardır. Bu iki çağırım arasındaki fark, CUDA dilinin GPU'lar üzerinde çalışacak olan fonksiyonların çağırılmadan önce konfigüre edilmelerinden kaynaklanmaktadır. Yapılan konfigürasyon sonrasında belirlenen mimariye göre çağırılan fonksiyon çalışırken, paralellik seviyesi bu aşamada belirlenmektedir.

CUDA dilinin bir başka özelliği ise donanıma yakın bir dil olmasından kaynaklanan, donanım bileşenlerine (GPU çekirdekleri, hafızalar ve yazmaçlar) kolay erişim imkanı sunmasıdır. Şekil 3.7'de (Ghorpade vd. 2012) CUDA işlemci modeli ve hafıza tipleri gösterilmiştir. CUDA işlemci modeli 3 ana bölümden oluşmaktadır: iş parçacıkları (*threads*), bloklar (*blocks*) ve ızgaralar (*grids*). En temel anlatımla bir blok içerisinde birden fazla iş parçacığı, bir ızgara içerisinde birden fazla blok ve bir GPU içerisinde birden fazla ızgara bulunmaktadır.



Şekil 3.7 CUDA işlemci modeli ve Hafıza tipleri (Ghorpade vd. 2012)

CUDA bileşenlerine ait temel kavramlar aşağıdaki şekilde anlatılabilir:

- **Ev sahibi (Host):** GPU'yu yapılandıran, GPU fonksiyonlarını çağırıp kullanan işlemcidir. Bir diğer deyişle CPU'nun kendisidir.

- **Cihaz (Device):** Paralelleştirme işleminin başladığı yerdir. Yani, GPU'nun kendisidir.

- **Çekirdek (Kernel):** Cihaz üzerinde koşan fonksiyondur. Paralleştirme işlemi yapıldığı anda, yazılmış olan fonksiyon ayrı çekirdekler üzerinde çalıştığından, her bir çekirdek üzerinde koşan fonksiyon olarak da adlandırılabilir.

- **Izgara:** Aynı çekirdek üzerinde koşan asenkron iş parçacığı grubu olarak adlandırılır. CPU tarafından kullanılan her GPU fonksiyonu ızgara üzerinden çağırılır. Izgaralar bir GPU'ya aittir ve multi-GPU sistemlerde, ızgaralar GPU'lar arasında paylaşılamaz.

- **Blok:** Izgaraları oluşturan, iş parçacıklarının ve paylaşımlı hafızanın oluşturduğu mantıksal bir birimdir. Bloklar bir ızgaraya, dolayısıyla bir GPU'ya aittir ve GPU'lar arasında paylaşılamaz. Blok yerleşimi 1B veya 2B olarak yapılabilir. Bir GPU içerisinde en fazla 65535 blok bulunabilir.

- **İş parçacığı:** Blokları oluşturan en küçük birimdir. Aynı zamanda GPU'nun temel birimidir ve CUDA içerisinde yazılan kodlar iş parçacıkları içinde çalışır. Blok içerisinde iş parçacığı yerleşimi 1B, 2B veya 3B olarak yapılabilir. GPU modeline göre değişebilmekle beraber, bir blok içerisinde en fazla 512 veya 1024 iş parçacığı bulunabilir.

CUDA işlemci modelinde bulunan hafıza tipleri aşağıdaki gibidir:

- **Global Hafıza (Global Memory):** GPU üzerindeki en genel yazılabilen ve okunabilen hafıza tipidir. Önbellekleme yapılmayan global hafıza oldukça yavaştır. Yaşam süresi GPU'nun yaşam süresi ile aynıdır. Hem CPU hem de GPU tarafından okuma/ yazma amaçlı kullanılır.

- **Doku Hafıza (Texture Memory):** Sadece okuma amaçlı kullanılan hafızadır. 2B erişim için önbellekleme yapılan doku hafızada, az güncellenen ama uygulama içinde çoğunlukla ihtiyaç duyulan veriler bulunur. Yaşam süresi GPU'nun yaşam süresi ile aynıdır.

- **Sabit Hafıza (Constant Memory):** Uygulama boyunca kullanılan sabit değerlerin ve çekirdek argümanlarının bulunduğu hafızadır. Önbellekleme yapılan sabit hafıza sadece okuma amaçlı kullanılır ve iş parçacıkları tarafından doğru bir şekilde önbellekli sabit hafızaya erişildiğinde bir yazmaç kadar hızlıdır. Yaşam süresi GPU'nun yaşam süresi ile aynıdır. Sabit hafıza, global hafıza gibi hem CPU hem de GPU tarafından kullanılır.

- **Paylaşımlı hafıza (Shared Memory):** Bir blok içinde tanımlı olan hafızadır. Blok içerisindeki tüm iş parçacıkları paylaşımlı hafızaya ulaşabilir. Okunabilir/ yazılabilir bir hafıza olan paylaşımlı hafıza, global hafızaya göre çok daha küçük boyutta ama çok daha hızlıdır. Paralellik seviyesi belirlenirken, bir blok

içerisindeki eş zamanlı çalışacak olan iş parçacıklarından, iş parçacığı başına ayrılan paylaşımlı hafıza boyutu dikkate alınır. Paylaşımlı hafıza boyutu değişken olup, CPU tarafından paralelleştirme yapılırken kullanılan notasyon ile blok başına ayrılacak olan paylaşımlı hafıza boyutu belirlenir. Yaşam süresi bir bloğun yaşam süresi ile aynıdır. Sadece GPU tarafından erişilebilir.

- **Yerel Hafıza (Local Memory):** Okunabilir/ yazılabilir olan yerel hafıza sadece tek bir iş parçacığı tarafından erişilebilir durumdadır. Yaşam süresi bir iş parçacığının yaşam süresi ile aynıdır. Çoğunlukla yazmaçların kullanılamadığı durumlarda kullanılan, önbelleği olmayan oldukça yavaş bir hafızadır.

- **Yazmaç (Register):** Sadece bir iş parçacığı tarafından erişilebilir olan yazmaç okunabilir/ yazılabilir en hızlı hafızadır. Yaşam süresi bir iş parçacığının yaşam süresi ile aynıdır. Bir iş parçacığına birden fazla yazmaç atanabilir.

CUDA dilinde yazılan bir kodda hangi fonksiyonların veya değişkenlerin CPU, hangilerinin GPU üzerinde koşacağını belirtmek, CUDA işlemci modelinde bulunan hafızalara erişebilmek ve paralelleştirme konfigürasyonları yapıldıktan sonra oluşturulan birden fazla ızgara, blok ve iş parçacıklarından hangisi üzerinde fonksiyonun koştuğunu hesaplayabilmek için niteleyiciler ve yerleşik değişkenler kullanılır. CUDA diline özgü olan bu niteleyiciler ve yerleşik değişkenler, donanım (GPU, CPU) ve yazılım arasında birer köprü görevi görür.

Niteleyiciler, fonksiyonlar ve değişkenler için olmak üzere iki farklı kullanıma sahiptir. Fonksiyon niteleyicileri bir fonksiyonun hangi işlemci üzerinde koşacağını belirtir ve aşağıdaki gibidir (Anonim 2009):

- **__device__ :** Fonksiyonun GPU üzerinde koşacağını belirtir. Sadece GPU fonksiyonları tarafından çağırılabilir.

- **__global__ :** Fonksiyonun GPU üzerinde koşacağını belirtir. Sadece CPU fonksiyonları tarafından çağırılabilir. CPU'dan GPU'ya geçişin ilk adımıdır. __global__ niteleyicisi ile başlayan fonksiyonlar mutlaka "void" dönüş tipinde olmalıdır.

- **__host__ :** Fonksiyonun CPU üzerinde koşacağını belirtir. Sadece CPU fonksiyonları tarafından çağırılabilir. Bir fonksiyonun CPU üzerinde koşacağını belirtmek için __host__ niteleyicisi kullanmanın yanı sıra, __host__ , __device__ ya da __global__ niteleyicilerinden hiç birinin kullanılmaması da yeterlidir.

- __host __ ile __device__ niteleyicileri birlikte kullanıldığında o fonksiyon hem CPU hem de GPU fonksiyonu haline gelir.

Değişken niteleyicileri bir değişkenin hafızadaki yerini belirtir ve aşağıdaki gibidir (Anonim 2009):

- **__device__ :** Değişkenin global hafızada tutulduğunu belirtir.Aynı ızgaradaki bütün iş parçacıkları tarafından erişilebilir durumda olan bu değişkenin yaşam süresi GPU'nun yaşam süresi ile aynıdır. __local__, __constant__ ve __shared__ niteleyicilerinden önce __device__ değişkeni opsiyonel olarak kullanılabilir.

- **__local__ :** Değişkenin lokal hafızada tutulduğunu belirtir. Sadece bir iş parçacığı tarafından erişilebilir durumda olan bu değişkenin yaşam süresi de bir iş parçacığının yaşam süresi ile aynıdır.

- **__constant__ :** Değişkenin sabit hafızada tutulduğunu belirtir. Aynı ızgaradaki bütün iş parçacıkları tarafından erişilebilir durumda olan bu değişkenin yaşam süresi GPU'nun yaşam süresi ile aynıdır. Ayrıca, CPU fonksiyonları tarafından da erişilebilir.

- **__shared__ :** Değişkenin paylaşımlı hafızada tutulduğunu belirtir. Aynı blok içindeki bütün iş parçacıkları tarafından erişilebilir durumda olan bu değişkenin yaşam süresi o bloğun yaşam süresi ile aynıdır.

- Bir değişkenin tanımından önce herhangi bir CUDA niteleyicisi kullanılmadığı durumda o değişken genel hafızada tutulur. Aynı ızgaradaki bütün iş parçacıkları tarafından erişilebilir durumdadır ve yaşam süresi GPU'nun yaşam süresi ile aynıdır. Ayrıca bu değişken CPU fonksiyonları tarafından da erişilebilir.

Yerleşik değişkenler kullanılarak ızgara ve blok boyutları belirlenir ve blok ve iş parçacığı numaraları hesaplanır. Yerleşik değişkenler sadece GPU fonksiyonları tarafından çağırılabilir ve aşağıdaki gibidir (Anonim 2009):

- **gridDim :** Izgaranın boyutunu belirtir. Bloklar, ızgaralar içerisinde sadece 1B veya 2B olarak yerleştirilebilir. 1B yerleştirildiğinde gridDim.x, 2B yerleştirildiğinde gridDim.x ve gridDim.y değişkenleri kullanılır.

- **blockDim :** Blok boyutunu belirtir. İş parçacıkları, bloklar içerisinde 1B, 2B ve 3B olarak yerleştirilebilir. İş parçacıkları bloklar içerisine 1B olarak yerleştirildiğinde blockDim.x, 2B olarak yerleştirildiğinde blockDim.x ve blockDim.y ve 3B olarak yerleştirildiğinde blockDim.x, blockDim.y ve blockDim.z değişkenleri kullanılır.

- **blockIdx :** Herhangi bir bloğun ızgara içerisinde oluşturulmuş bloklar arasındaki yerini işaret eder.

- **threadIdx :** Herhangi bir iş parçacığının bir blok içerisinde oluşturulmuş iş parçacıkları arasındaki yerini işaret eder.

- **warpSize :** Bir blok içerisinde bir arada çalışan maksimum iş parçacığı sayısını belirtir. Bu iş parçacıkları aynı komutu işlediklerinde (kod içerisinde iş parçacığının numarasına göre if-else veya switch işlemi yapılmamışsa) ve hafıza erişimlerinde bir arada ve düzen içinde kullanıldıklarında (sabit hafıza da aynı değeri okuyor ise veya paylaşımlı hafıza da her bir iş parçacığı kendine ait bölgeyi kullanıyor ise) oldukça yüksek hızlarda çalışırlar. GPU modeline göre warpSize değişkeninin belirttiği değer değişmekte olup, paralelleştirme konfigürasyonları temelde bu değere göre yapılmaktadır.

Şekil 3.8'de ızgara, blok ve iş parçacıklarının 1B ve 2B örnek yerleşim düzenleri gösterilmiştir.

Şekil 3.8 1B ve 2B örnek blok ve iş parçacığı yerleşim düzeni

Şekil 3.8'de gösterilen 1B örnek yerleşim düzenine göre, bloklar ızgara içerisinde tek boyutlu olarak yerleştirilmiştir. Ayrıca her bir blok içerisinde 1B iş parçacığı yerleşimi mevcuttur. Bu düzene göre, bir bloğun yerini hesaplayabilmek için aşağıdaki formül kullanılır:

UniqueBlockIndex = blockIdx.x;

Bir iş parçacığının yerini hesaplayabilmek için de aşağıdaki formül kullanılır:

UniqueThreadIndex = blockIdx.x * blockDim.x + threadIdx.x;

Şekil 3.8'de gösterilen 2B örnek yerleşim düzenine göre, bloklar ızgara içerisinde iki boyutlu olarak yerleştirilmiştir. Ayrıca her bir blok içerisinde 2B iş parçacığı yerleşimi mevcuttur. Bu yerleşim düzenine göre, bir bloğun yerini hesaplayabilmek için aşağıdaki formül kullanılır:

UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;

Bir iş parçacığının yerini hesaplayabilmek için de aşağıdaki formül kullanılır:

UniqueThreadIndex = UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;

3.2 GPGPU

Merkezi İşlemci Birimi (*Central Processing Unit* - CPU) üzerinde yapılan hesaplamaların, bilgisayarın sadece grafik hesaplamalarını yapan GPU üzerinde yapılması Genel Amaçlı Grafik İşlemci Birimleri (GPGPU) olarak adlandırılmıştır. Tez kapsamında, farklı görevlere ait farklı kısıtlar için algoritmanın parametreleri değiştirilerek oluşturulan birden fazla Çok Amaçlı Optimizasyon Problemleri'nin çözümünde GPGPU'lar kullanılmıştır.

3.2.1 GPGPU'ların bilimsel hesaplamadaki yeri ve önemi

GPGPU'lar, geleneksel CPU'lara göre aynı miktarda yatırım ile 14-50 kat daha hızlı çözümler sağlayan maliyet etkin çözümler olmaları nedeniyle, özellikle son yıllarda bilimsel hesaplama camiasında büyük yankı uyandırmaktadır. Ayrıca, günümüzdeki GPU'lar sabit ve sınırlı bir komut seti yerine hayli gelişkin komut setleri ile geniş programlama imkânı sunmaktadır (Fernando ve Kilgard 2003, Fernando 2004). Bu özellikleri nedeniyle GPU'lar, son zamanlarda grafik uygulamalar haricindeki bir takım çalışmalarda da kullanım alanı bulmaktadır.

Thompson vd. (2002), bilim camiasının dikkatinin GPU'ların bilimsel hesaplama amacıyla da kullanılabilirliğine dair öncü bir çalışma olmuştur. Ardından Krüger ve Westermann (2003), lineer cebir operatörlerinin GPU'lar üzerinde uygulanabilirliğini göstermiş; Bolz vd. (2003) de GPU'lar üzerindeki ilk seyrek matris denklem çözümüne yönelik yazılım kütüphanelerini geliştirmiş; Hillesland vd. (2003) GPU'lar üzerinde optimizasyon algoritmalarının gerçeklenmesine dair ilk adımları atmış; Govindaraju vd. (2004) ise veri tabanı uygulamalarının GPU'lar üzerinde geliştirilmeleri durumunda ne kadar hızlı olabileceğini göstermiştir. Bütün bu öncü çalışmaların ardından, dünyanın çeşitli bölgelerindeki değişik araştırma grupları, bilimsel hesaplamalarda GPGPU'ların avantajlarından yararlanmaya yönelik çalışmalar yapmaya başlamıştır.

Günümüzde, üst seviye yazılım geliştirme platformları ve Uygulama Geliştirme Arayüzleri (Application Programming Interfaces - APIs) (örneğin CUDA veya OpenCL), GPU mimarileri üzerinde optimum performansta çalışmaya yönelik yazılım geliştirilmesine de olanak sağlamaktadır.

3.2.2 GPGPU'lar üzerinde buluşsal algoritmalar

Genetik Algoritma ve türevlerinin GPU mimarileri üzerinde gerçeklenmesine ilişkin çalışmalar, 2005 yılında başlamıştır. Bu konudaki ilk çalışmalar, daha ziyade kavramsal ispat amaçlı makaleler olup; GPU'ların hızlı çalışan algoritmalar için ne kadar uygun olduğunu göstermeye yönelik olmuştur. Bu çalışmalar, kronolojik sıra ile (Yu vd. 2005, Wong vd. 2005, Wong ve Wong 2006, Fok vd. 2007, Wong ve Wong 2009, Tsutsui ve Fujimoto 2009, Debattisti vd. 2009, Pospichal 2010, Pospichal vd. 2010a, Pospichal vd. 2010b, Kannan ve Ganji 2010, Krömer vd. 2011) olarak özetlenebilir. Söz konusu çalışmalar genel olarak konvansiyonel Genetik Algoritma'nın bir takım kıyaslama problemleri üzerindeki uygulamaları üzerine olup; bunlardan sadece birinde (Tsutsui ve Fujimoto 2009) Genetik Algoritma'nın kuadratik atama problem üzerindeki bir uygulaması GPGPU'lar üzerinde gerçekleştirilmiştir. Bu çalışmada, GPGPU üzerinde CPU implementasyonuna kıyasla 10 kata varan hızda bir başarım sağlandığı belirtilmiştir.

Parçacık Sürüsü Optimizasyonu (PSO)'nun GPU mimarileri üzerinde gerçeklenmesine dair ilk çalışmalar da, kavramsal ispat amaçlı makaleler olup; GPU'nun PSO için ne kadar uygun olduğunu göstermeye yönelik olmuştur. Li vd. (2007), çok işlemcili ortamlarda yoğun veri iletişimi gerektiren büyük popülasyonların ele alınması için GPU mimarilerinin uygun olduğunu göstermişlerdir. GPU'lardaki "Tek Komut, Çoklu İşlem Akışı (*Single Instruction Multiple Thread* – SIMD)" mekanizmasının ve aynı anda erişilebilir ortak hafıza mimarisinin, PSO'da çok etkin bir şekilde kullanılabilecek bir özellik olduğunu belirtmişlerdir.

Zhou ve Tan (2009), 2007 tarihli PSO algoritmasını (Bratton ve Kennedy (2007) tarafından önerilmiş olan ve Sürü Zekâsı camiası tarafından "Standart Parçacık Sürüsü Optimizasyonu - SPSO" olarak kabul edilen algoritma versiyonu) GPU'lar üzerinde gerçekleştirerek; CPU üzerinde çalışan versiyona göre 11 kata varan hız artışı gözlemlemişlerdir.

Mussi ve Cagnoni (2009), algoritma içerisindeki dahili verileri (parçacık durum vektörleri, lokal ve global en iyi pozisyonlar, vb.), hafızadan okuma / hafızaya yazma işlemlerini en aza indirgeyecek şekilde yapılandırmışlardır. Bu sayede, 3 alt-sürünün kullanıldığı 100 boyutlu problemlerde, normal CPU versiyonuna göre 100 kat hız artışı gözlemlemişlerdir.

Literatürdeki iki çalışmada ise, çalışma zamanı içerisinde PSO'da binlerce defa tekrarlanan rastgele sayı üretimi işlemi üzerine odaklanılmıştır. Mussi vd. (2009), rastgele sayı üretme işlemini GPU üzerinde bulunan "*Mersenne Twister*" kullanarak (hiç CPU kullanmadan) gerçekleştirmiştir. Bastos-Filho vd. (2010) ise aynı amaçla "*Linear Congruential Generator*" ve "*George Marsaglia Generator*" tekniklerini kullanmışlardır. Her iki çalışmada da, bu yöntemler ile rastgele sayı üretimi aşamasında GPU-CPU haberleşmesi ihtiyacının ortadan kalkması dolayısıyla önemli hız artışı gözlemlenmiştir. Önerilen rastgele sayı üreteçlerinin çıktılarının istatistiksel özelliklerinin de yeterli kalitede çözüm elde edilmesini sağlayacak şekilde olduğu belirtilmiştir.

de P. Veronese ve Krohling (2009), aslen temel olarak C programlama diline dayanan ve GPU çekirdeklerine özgü bir takım komut setlerinin eklenmiş olduğu C-CUDA (*C – Compute Unified Device Architecture*) programlama dili ile geliştirdikleri PSO algoritması ile, özellikle büyük sürülerin kullanıldığı çok boyutlu problemlerde orijinal C programlama dili ve MATLAB'a göre 10 kat daha hızlı bir şekilde çözümler elde etmişlerdir.

Solomon vd. (2011), GİBAP projesi kapsamında ele alınacak olan görev atama probleminin çözümüne yönelik olarak "çok sürülü" bir PSO algoritmasını GPU'lar üzerinde geliştirmişlerdir. Bu çalışmada da çok-sürülü ve paralel GPU PSO versiyonunun, özellikle çok yüksek boyutlu problemlerde sıralı çalışan PSO'ya göre 37 kat daha hızlı olduğu gözlemlenmiştir.

Cardenas-Montes vd. (2011), özellikle çok çok yüksek boyutlu (5000-15000 mertebesinde) problemlere odaklanmışlar; uygunluk fonksiyonu değerinin hesaplanması adımında her bir boyuttaki hesaplamaları GPU'daki farklı çekirdeklere dağıtarak algoritmayı paralelleştirmişlerdir. 15000 boyutlu problemler için, CPU'lara göre 20 kata varan hız artışları gözlemlemişlerdir.

Castro-Liera vd. (2011), GPU üzerinde başarılı bir PSO versiyonu daha geliştirmişlerdir. Alt-sürülerin boyutlarını ayarlayarak ve bu alt-sürüleri GPU içerisindeki bloklara dağıtarak, tek GPU ile, 8-CPU'lu bir makineye eşdeğer performans elde edilebileceğini göstermişlerdir.

Mussi vd. (2011), PSO'da paralelizasyon sağlamak için iki temel yaklaşım olabileceğini belirtmişlerdir:

- Parçacık hareketlerindeki senkronizasyonun tamamen bozulmasına yönelik olan ve daha kolay/ aşikâr olan yaklaşım,
- "Ada modeli" olarak da adlandırılan, alt-sürüler arasında minimal veri değişimi esasına dayanan daha karmaşık olan yaklaşım

Mussi vd. (2011), GPU'lar üzerinde, her iki yaklaşım uyarınca iki farklı paralel-PSO algoritması geliştirmişlerdir. Bir versiyonda, her bir alt-sürüyü tek bir iş parçacığı içerisinde koşturmuş; diğer versiyonda ise alt-sürüler arasındaki koordinasyonu global hafıza üzerinden gerçekleştirmişlerdir. Her iki yöntemin de avantaj ve kısıtlarını değerlendirmeye çalışmışlardır. İlk yöntemde, alt-sürü sayısının (her bir alt-sürüdeki parçacık sayısının) teorik üst sınırını hesaplamışlardır; bu sayıların donanımın kabiliyetleri ile doğrudan ilintili olduğunu göstermişlerdir. İkinci yöntemin faydalarının ise ancak çok boyutlu problemlerde gözlemlenebildiğini; düşük boyutlu problemlerde ise yöntemin ekstra maliyetlerinin getirilerinden daha fazla olduğunu belirtmişlerdir.

Zhou ve Tan (2011), GPU'lar üzerindeki ilk çok-amaçlı PSO algoritmasını geliştirmişlerdir. Söz konusu çalışma, her bir amaç için ayrı bir alt-sürü koşturulması, her bir iterasyon sonunda alt-sürüler arasında birey değişimi yapılması esasına dayanmaktadır. Bu işlemlerin CPU'lar yerine GPU'larda yapılmasının 3,74-7,92 kat hız artışı sağladığını göstermişlerdir.

Çok yakın zamanda Hung ve Wang (2012), 100 boyutlu bir yük dengeleme probleminde, NVIDIA Tesla C1060 1.30 GHz GPU kartı üzerinde yapılan işlemlerin tek-çekirdekli Intel Xeon-X5450 3.00 GHz CPU kartına göre 280 kat; çift çekirdekli Intel Xeon-X5450 3.00 GHz CPU kartına göre ise 83 kat daha hızlı olduğunu hesaplamışlardır.

Tanımı ve doğası gereği, Görev Planlama Problemi'ne uyarlanması en uygun olan algoritmalardan bir diğeri de, Karınca Kolonisi Optimizasyon Algoritması'dır (*Ant Colony Optimization* - ACO). Karınca Kolonisi Optimizasyonu Algoritması'nın (ACO) GPU'lar üzerinde gerçeklenmesine ilişkin çalışmalar, 2007 yılı sonrasında hız kazanmıştır. Catala vd. (2007), farklı yaklaşımlar ile tanımladıkları 3 farklı ACO versiyonunu GPU'lar üzerinde gerçekleştirmişler; bu üç versiyonu birbirleri ile karşılaştırmışlardır. Her üç versiyonun da çok yüksek boyutlu (örneğin 32000) Orienteering Problemi'ni çok kısa sürelerde çözebildiğini göstermişlerdir.

You (2009), Gezgin Satıcı tarzı bir problemde, GPU üzerindeki herhangi bir paralel ACO implementasyonunun verimliliğinin, özellikle problem boyutu arttığında ön plana çıktığını göstermiştir. Buna göre CUDA'da geliştirilen paralel GPU implementasyonu, C'de geliştirilen CPU implementasyonuna göre:

- 60 boyutlu (yani 60 şehirli) problemde 2 kat hızlı iken,
- 400 şehirli problemde 12 kat,
- 800 şehirli problemde ise 23 kat

hızlı olabilmektedir.

Baia vd. (2009), ACO'nun MAX-MIN Karınca Sistemi (MAX-MIN Ant System - MMAS) olarak da bilinen versiyonunu CUDA ile gerçekleştirmişler; Gezgin Satıcı Problemi'nin çözümünde normal CPU implementasyonuna oranla 32 kat daha yüksek hız performansı elde etmişlerdir.

Fu vd. (2010), MATLAB için geliştirilmiş ve .MEX dosyalarının içeriğinin CPU üzerinden GPU'lara aktarılması prensibi ile çalışan Jacket araç kutusunu kullanarak bir ACO versiyonu geliştirmişlerdir. Kavramsal ispat amaçlı bu çalışmada, normal CPU implementasyonu ile herhangi bir performans kıyaslaması yapılmamış/ sunulmamıştır.

Tsutsui ve Fujimoto (2011), ACO'yu Kuadratik Atama Problemi'nin çözümü için uyarlamışlar; CUDA ile geliştirilen GPU versiyonunun, normal CPU versiyonuna göre 24.6 kat daha hızlı çalıştığını belirtmişlerdir.

Cecilia vd. (2011), ACO algoritmasının 3 aşamasının (tur oluşturma, feromon güncelleme, rulet çarkı tabanlı seçim) ayrı ayrı paralelleştirilebileceğini belirtmişler; tur oluşturma aşamasının GPU'larda paralelleştirilmesi ile 28 kat, feromon güncelleme aşamasının paralelleştirilmesi ile ise 20 kata varan hız iyileştirmesinin mümkün olabileceğini göstermişlerdir. Çok yakın tarihli bir çalışmada yine Cecilia vd. (2013) tarafından, ACO içerisindeki hem tur oluşturma, hem feromon güncelleme, hem de rulet çarkı tabanlı seçim işlemlerinin tamamını paralelleştirmeye yönelik bir yaklaşım

sunulmuş; Gezgin Satıcı Problemi'nin çözümünde, CPU'lara oranla 20 kat daha hızlı bir GPU implementasyonu gerçekleştirilmiştir.

Yine çok yakın tarihli bir çalışmada Delevacq vd. (2013), ACO'nun MMAS versiyonunu Fermi GPU mimarileri üzerinde gerçekleştirmiş, Gezgin Satıcı Problemi'nin çözümünde CPU implementasyonuna oranla 23.6 kata varan hız performans artışı gözlemlemişlerdir.

4. ARAŞTIRMA BULGULARI

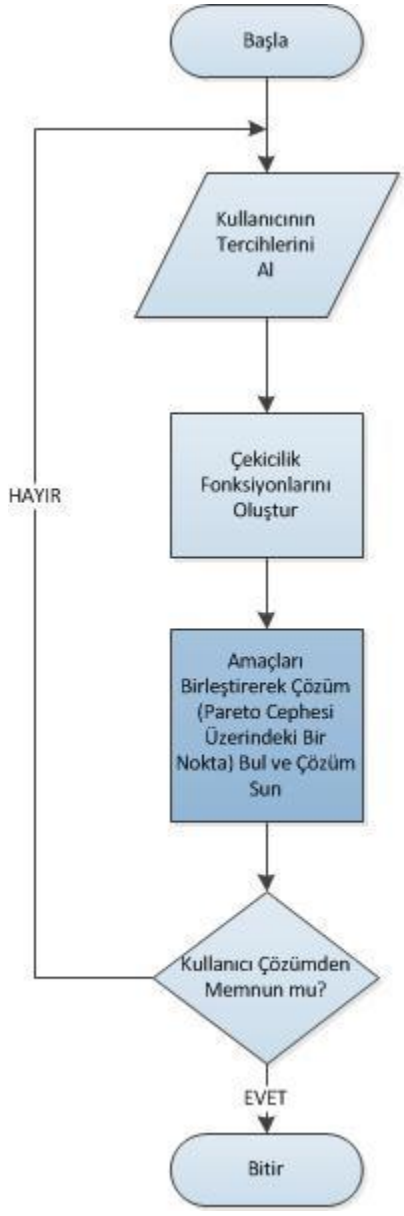
Çok Amaçlı Optimizasyon Problemi'nin paralelleştirme yöntemleri ile çözümüne yönelik atılacak olan ilk adım, problemin paralelleştirmeye uygun olup olmadığını belirlemektir. Eğer bir problem paralelleştirmeye uygun değil ise o probleme paralelleştirme yöntemleri uygulamak çözümü kolaylaştırmaz, aksine çözümü imkansız hale getirebilir. Paralelleştirme yöntemleri kullanılarak çözülebileceğine karar verilen problem için uygulanacak olan bir sonraki adım problemin çözümünün hangi aşamalarda paralelleştirileceğini belirlemek ve paralel çözümün başlangıç ve bitiş noktalarına (yani CPU'dan GPU'ya ve GPU'dan CPU'ya geçiş noktaları) ve paralel yöntemler ile elde edilen sonuçların CPU'da ne şekilde birleştirileceğine karar vermektedir.

Bu bölümde, yukarıda anlatılanlar bütünüyle ele alınarak tasarlanan mimari anlatılacak, implementasyon detaylı bir şekilde ele alınacak ve problemin çözümüne ait sonuçlar gösterilecektir.

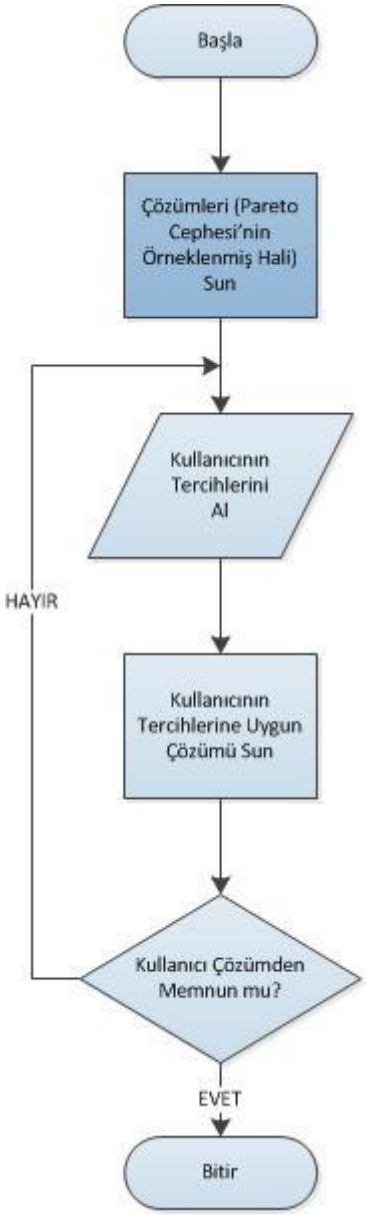
4.1 Mimari Yaklaşım

Çok Amaçlı Optimizasyon Problemleri'nin çözümünde kullanılan Seri Programlama mantığına dayalı geleneksel yöntemlerde, kullanıcının tercihlerine istinaden optimizasyon süreci (Şekil 4.1'de, koyu renkli kutucuk ile gösterilmektedir) başlatılmakta ve çözüm önerisi sunulmaktadır (Sel 2013). Kullanıcının bu çözümden memnun kalmaması durumunda, tercihlerin yeniden sorgulanması ve memnuniyet sağlanana değin optimizasyon sürecinin defalarca tekrarlanması gerekebilir. Bu da en uygun çözüme ulaşana kadar geçen sürenin oldukça uzun olması anlamına gelmektedir. Özellikle Çoklu Hava Unsurları için Görev Planlama Sistemleri gibi zaman kısıtı olan uygulamalarda, bir diğer deyişle belirli bir zamanda birden fazla unsurun görevinin en etkin bir biçimde planlanması gereken, aynı zamanda planlama yapılırken tüm amaçların yapılmış olması şartı gözetilen uygulamalarda bu yaklaşım yavaş kalmaktadır. Hedef ise, daha kısa zamanda en uygun çözümün bulunup uygulamaya konulmasıdır.

Öte yandan tez kapsamında önerilen sistem ile, tek bir optimizasyon süreci sonunda bütün olası optimum çözümlerin kullanıcıya sunulması; kullanıcının, tercihleri uyarınca bunlardan bir tanesini seçebilmesi mümkün hale gelmiştir (Şekil 4.2). Kullanıcının tercihlerini değiştirmesi, yeni bir optimizasyon sürecini gerektirmemekte; böyle bir durumda eldeki çözüm kümesinden yeni bir eleman kullanıcıya doğrudan sunulmaktadır. Bu sayede kullanıcının tercih farklılığına göre yeni bir çözüm elde etmek için ekstra süre harcanmamaktadır.



Şekil 4.1 Çok Amaçlı Optimizasyon Problemleri'nin çözümünde uygulanan Seri Programlama yaklaşımı



Şekil 4.2 Çok Amaçlı Optimizasyon Problemleri'nin çözümünde tez kapsamında uygulanan Paralel Programlama yaklaşımı

Ana hatları şekil 4.2'de gösterildiği gibi tasarlanan uygulama mimarisi ağırlıklı olarak GPU üzerinde çalışmaktadır ancak uygulama parametreleri ve kullanıcı tercihlerinin alınması ile sonuçların kullanıcıya gösterilmesi aşamaları CPU üzerinde çalışmaktadır. Uygulama, Microsoft Visual Studio 2008 ortamında geliştirilmiş ve Paralel Programlama kütüphanesi olarak NVIDIA CUDA 5.5 versiyonu kullanılmıştır. Grafik kartı modeli olarak NVIDIA Quadro K5000 seçilmiş ve özellikleri çizelge 4.1'de gösterildiği gibidir.

Çizelge 4.1 Geliştirme ortamı özellikleri

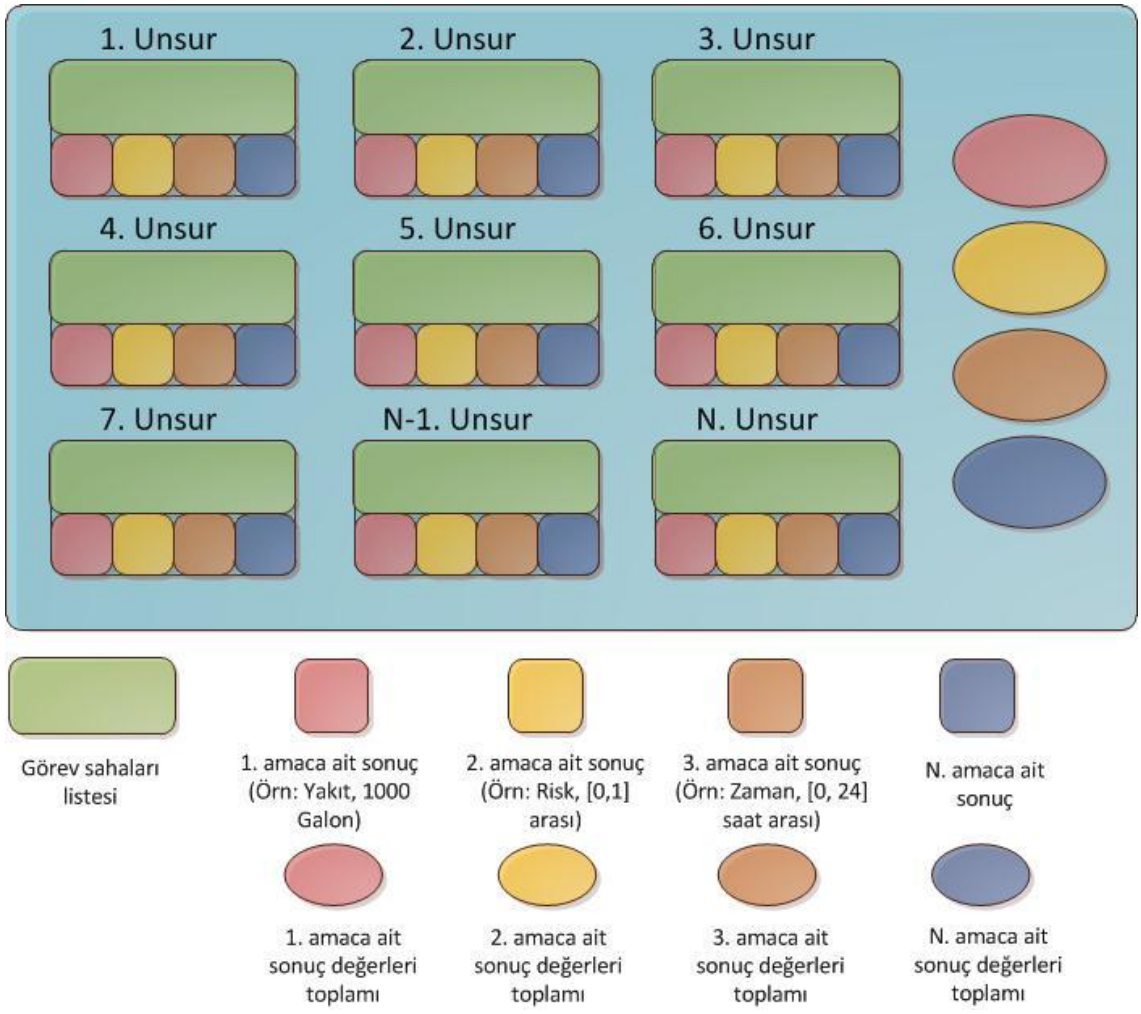
Cihaz Modeli	Quadro K5000
Genel Hafıza Boyutu	4GB
CUDA Çekirdekleri	1536
GPU ve Hafıza Saat Hızı	706 MHz ve 2700 MHz
Hafıza Veri Yolu Genişliği	256-bit
Çoklu İşlemci ve Blok için Maksimum İş Parçacığı Sayısı	2048 ve 1024
CUDA Sürücüsü ve Versiyonu	5.5 ve 3.0

Çok Amaçlı Optimizasyon Problemlerinin çözümünde uygulanabilecek yöntemlerden bazıları Sel (2013) tarafından kıyaslanmış ve Genetik Algoritmanın da bu tip problemlerinin çözümü için kullanılabilecek algoritmalardan biri olduğu gösterilmiştir. Bu tez çalışması kapsamında da, geliştirilen uygulamanın paralelleştirilen bölümünde, her bir iş parçacığı içerisinde buluşsal yöntemlerden biri olan seçici Genetik Algoritma kullanılarak sonuçlar elde edilmiştir. Seçici Genetik Algoritma'nın parametreleri çizelge 4.2'de gösterildiği gibi olup, aslında her bir iş parçacığı üzerinde koşan algoritmanın parametrelerini temsil etmektedir. Genetik Algoritma parametrelerinden en iyiyi seçme oranı, mutasyon oranı ve çaprazlama oranı değerleri de Sel (2013) çalışması sonunda elde edilen değerler olarak bu çalışmada kullanılmıştır.

Çizelge 4.2 Genetik Algoritma Parametreleri

Algoritma Parametresi	Değeri
Kromozom Büyüklüğü	16-bit
Popülasyon Büyüklüğü	100
Jenerasyon (İterasyon) Sayısı	100
En İyiyi Seçme Oranı	0.2
Mutasyon Oranı	0.1
Çaprazlama Oranı	0.9

Genetik Algoritma kavramlarından olan kromozom, bu çalışma için genleri ikili sayı sisteminde bir bit olan 16-bitlik diziyi temsil etmektedir. Gerçek kromozom yapısı şekil 4.3'te gösterildiği gibi olup, içerisinde N tane unsur ve görev sahası eşleşmesi bulunan rotalardan oluşan bir görevi temsil etmektedir ve asıl amaç en az yakıt harcanan ve görev riski minimum olan görevleri bulmaktır. Ancak yakıt, zaman ve risk hesabı tez çalışmasının amacı dışında olduğundan gerçek yakıt, zaman ve risk hesapları yerine, ilerleyen bölümlerde anlatılacak olan temsili hesaplamalar kullanılmıştır.



Şekil 4.3 Gerçek kromozom yapısı

Uygulama geliştirilme aşamasında öncelikle, yazılımın sadece üzerinde bulunduğu grafik kartı ile değil, eski ve yeni tüm grafik kartları ile uyumla çalışabilir olmasına dikkat edilmiş ve bunun için üzerindeki grafik kartının özelliklerini okuyup ona göre iş

parçacığı ve blok sayısını hesaplayan kodun implementasyonu yapılmıştır. Blok ve ızgara boyutları kullanıcıdan alınan giriş parametrelerine göre grafik kartının uygunluğu da dikkate alınarak hesaplanmıştır. Örnek hesaplama algoritması aşağıdaki gibi çalışmaktadır:

- Kullanıcıdan gelen "uygulama sonunda ekranda gösterilecek sonuç sayısı" değerini al ve mevcut grafik kartı üzerinde istenen sayıda sonuç için paralelleştirme yapılıp yapılamayacağını kontrol et.

- Eğer mevcut grafik kartı istenilen sayıda sonuç için paralelleştirme işlemine uygun değil ise kullanıcıya uyarı mesajı göster ve yeni bir değer girmesini bekle.

- Kullanıcının yeni girdiği değeri al ve 1. işlemi uygula. Eğer kullanıcıdan gelen sonuç sayısı uygun ise uygulama konfigürasyon dosyasında bulunan "her bir blok için maksimum iş parçacığı sayısı" değerini al.

- Grafik kartı özelliklerinden, eş zamanlı olarak birlikte çalışabilen maksimum iş parçacığı sayısı anlamına gelen "warpsize" değerini oku ve girilen sonuç sayısına göre birlikte çalışabilen iş parçacığı grubu (warpPerBlock) sayısını hesapla. Daha sonra her bir blok içinde kaç iş parçacığı grubu olacağını hesapla.

- Her bir blok içindeki iş parçacığı sayısını, warpsize değeri ve iş parçacığı grubu sayısı çarpımı ile hesapla.

- Toplam istenilen sonuç sayısını, her bir blok içinde çalışacak iş parçacığı sayısına bölerek blok sayısını hesapla.

Yukarıda anlatılan işlemlere ait sözde kod Şekil 4.4'te gösterildiği gibidir. Her bir blok için maksimum iş parçacığı sayısı, grafik kartının desteklediği maksimum iş parçacığı değerinden farklı olarak uygulamanın geri uyumlu ve daha verimli çalışabilmesi için belirlenen bir değerdir. Mevcut grafik kartı için desteklenen değer 1024 iken, uygulama konfigürasyon dosyasına yazılan değer 256'dır. Eldeki grafik kartının oldukça ileri model bir donanım olduğu ve uygulamanın kullanılacağı diğer bilgisayarda ortalama standartlarda bir grafik kartı bulunduğu düşünüldüğünde 256 değeri hemen her grafik

kartı için uygun bir değer olduğu görülmüştür. Her bir blok için maksimum iş parçacığı sayısı hesaplanırken dikkate alınan bir diğer durum ise bir blok içinde ne kadar fazla iş parçacığı bulunursa, her bir iş parçacığı başına düşen paylaşımlı hafıza alanının o kadar az olmasıdır. Dinamik paylaşımlı hafıza tahsis işleminde, blok başına düşen paylaşımlı hafıza alanı tahsis edilmektedir. Tahsis edilen toplam paylaşımlı hafıza alanı, o blok içinde bulunan iş parçacığı sayısına bölünerek iş parçacığı başına düşen paylaşımlı hafıza alanı bulunur. Sonuç olarak, bir blok içinde bulunan iş parçacığı sayısı, iş parçacığı başına kullanılabilir paylaşımlı hafıza alanı ile ters orantılıdır. Bu yüzden paralelleştirme miktarı (iş parçacığı sayısı) hesaplanırken yukarıda anlatılanlar gibi parametreler göz önünde bulundurulmuştur.

```
GPU özelliklerini oku;
Kullanılabilecek maksimum iş parçacığı sayısını hesapla;
Kullanıcı tarafından girilen istenilen sonuç sayısı değerini al;

IF kullanıcı tarafından girilen istenilen sonuç sayısı değeri > kullanılabilecek maksimum
iş parçacığı sayısı
    Kullanılabilecek iş parçacığı sayısını -1 olarak ayarla;
    Kullanılabilecek blok sayısını -1 olarak ayarla;
ELSE
    GPU özelliklerinden warpsize değerini al;
    Birlikte çalışabilen iş parçacığı grubu sayısını hesapla;
    Her bir blok içerisinde kaç adet iş parçacığı grubu olacağını hesapla;
    Toplam blok sayısını hesapla;
    Hesaplanan iş parçacığı sayısını kullanılabilecek iş parçacığı sayısı olarak ayarla;
    Hesaplanan blok sayısını kullanılabilecek blok sayısı olarak ayarla;
ENDIF
```

Şekil 4.4 Izgara ve blok boyutlarının hesaplanmasını gösteren sözde kod

Mevcut grafik kartının desteklediği en uygun ızgara ve blok boyutları hesaplandıktan sonra her bir iş parçacığı için ilk popülasyonların oluşturulması işlemine geçilmiştir. Kullanıcı tarafından girilen sonuç sayısı kadar popülasyonlar için öncelikle CPU üzerinde gerekli miktarda hafıza tahsis edilmektedir. Her bir popülasyon içindeki tüm kromozomlar yaratılıp popülasyona eklendikten sonra, CPU üzerinde popülasyonlar yaratılmış olur.

CPU üzerinde yaratılmış olan popülasyonlara rastgele başlangıç değerleri atama işleminin paralel olarak yapılabilmesi için popülasyonların öncelikle GPU'ya aktarılması gerekmektedir. Bu işlem oldukça karmaşık ve dikkatle ele alınması gereken bir işlem olduğundan, bu esnada yapılan en ufak bir hata uygulamanın çalışmasını

engellemektedir. Çünkü GPU üzerinde programlama seviyesi henüz CPU üzerindeki programlama seviyesine ulaşmamıştır. Dolayısıyla nesne yönelimli programlama mantığı GPU üzerinde tam olarak uygulanamamaktadır. Bu yüzden çalışma boyunca yüksek seviye nesneler yerine düşük seviye, çoğunlukla dizilerden oluşan yapılar kullanılmıştır. Buna göre bir popülasyon yapısı içerisinde, kromozom yapısı dizisinin işaretçisi ile popülasyon büyüklüğü değeri, bir kromozom yapısı içerisinde de gen ve uygunluk değerleri dizileri ile kromozom büyüklüğü değeri bulunmaktadır.

Bu yapıya göre, CPU'da oluşturulan tüm popülasyonların GPU'ya aktarılması aşağıdaki gibi olmaktadır:

- Kromozom ve popülasyon büyüklükleri alınıp, sonuç sayısı ve popülasyon büyüklüğü çarpımı ile toplam büyüklük değeri bulunur.

- Bir kromozom yapısının hafızada işgal ettiği alan ve toplam popülasyon büyüklüğü çarpımı ile toplam kromozom yapısının hafıza büyüklüğü hesaplanır ve GPU'da bu alan kromozom dizisinin işaretçisi için tahsis edilir. Bu işlem GPU üzerinde "cudaMalloc" fonksiyonu kullanılarak yapılmaktadır. Bu fonksiyon sadece GPU üzerinde istenilen büyüklükte hafıza tahsis etmek için kullanılır.

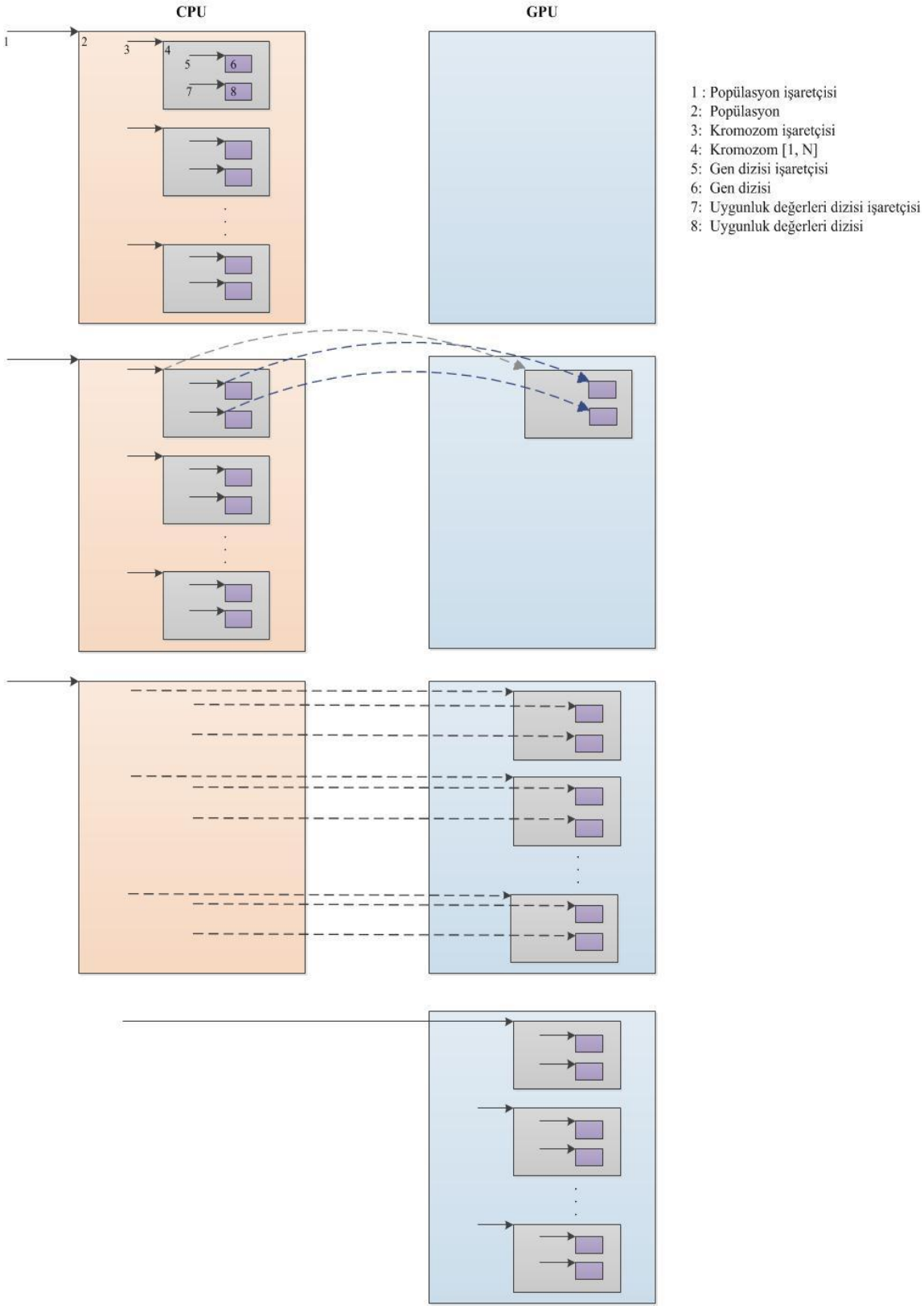
- Kromozom yapısında bulunan gen ve uygunluk değerleri dizisi için GPU üzerinde diziler yaratılır. Ardından GPU üzerinde yaratılan bu diziler CPU üzerindeki kromozom yapısında bulunan gen ve uygunluk değerleri dizilerine art arda atanır. Daha sonra aynı işlem kromozom yapısı için de yapılır, yani GPU üzerinde kromozom yapısı için bir kromozom büyüklüğünde hafıza tahsis edilir. Son olarak, popülasyon yapısı içinde bulunan ve kromozomların tümünü barındıran kromozom dizisi içinden ilgili kromozom işaretçisine GPU üzerinde yaratılan kromozom dizisi atanır. Bu işlem toplam kromozom sayısı kadar yinelenir ve böylece tüm kromozom dizisi GPU tarafına aktarılmış olur.

- Son olarak, CPU üzerinde kromozom dizisine işaret eden işaretçi de GPU tarafına aktarılır. Bu işlem için CUDA fonksiyonu olan "cudaMemcpy" kullanılmaktadır.

Yukarıda anlatılan adımların resimsel gösterimi şekil 4.5'deki gibidir.

Bütün popülasyonlar GPU'ya aktarıldıktan sonra popülasyonlardaki kromozomların her birine başlangıç değerleri atanması gerekmektedir. Başlangıç değerlerinin en büyük özelliği, bu değerlerin rastgele atanmasından dolayı her bir popülasyonun ilk durumunun diğer popülasyonlardan farklı olmasıdır. Bu da genetik algoritmaya mümkün olduğunca farklı sonuçlar elde etme imkanı vermektedir. Bir diğer deyişle, popülasyondaki genetik çeşitlilik artmaktadır.

Rastgele sayı üretme işlemi bütünüyle GPU üzerinde yapılmaktadır ve bu işlem için NVIDIA CUDA kütüphanesi olan Rastgele Sayı Üretme (CURAND- CUDA Random Number Generation) kütüphanesi kullanılmıştır. Bu kütüphane ile bir yandan oldukça yüksek performans ile rastgele sayı üretilebilirken, diğer yandan bu işlem doğrudan GPU üzerinde yapıldığından işlemciler arası veri geçişi yapılmasına gerek kalmamaktadır. Bu durum da uygulamanın performansını artırmaktadır.

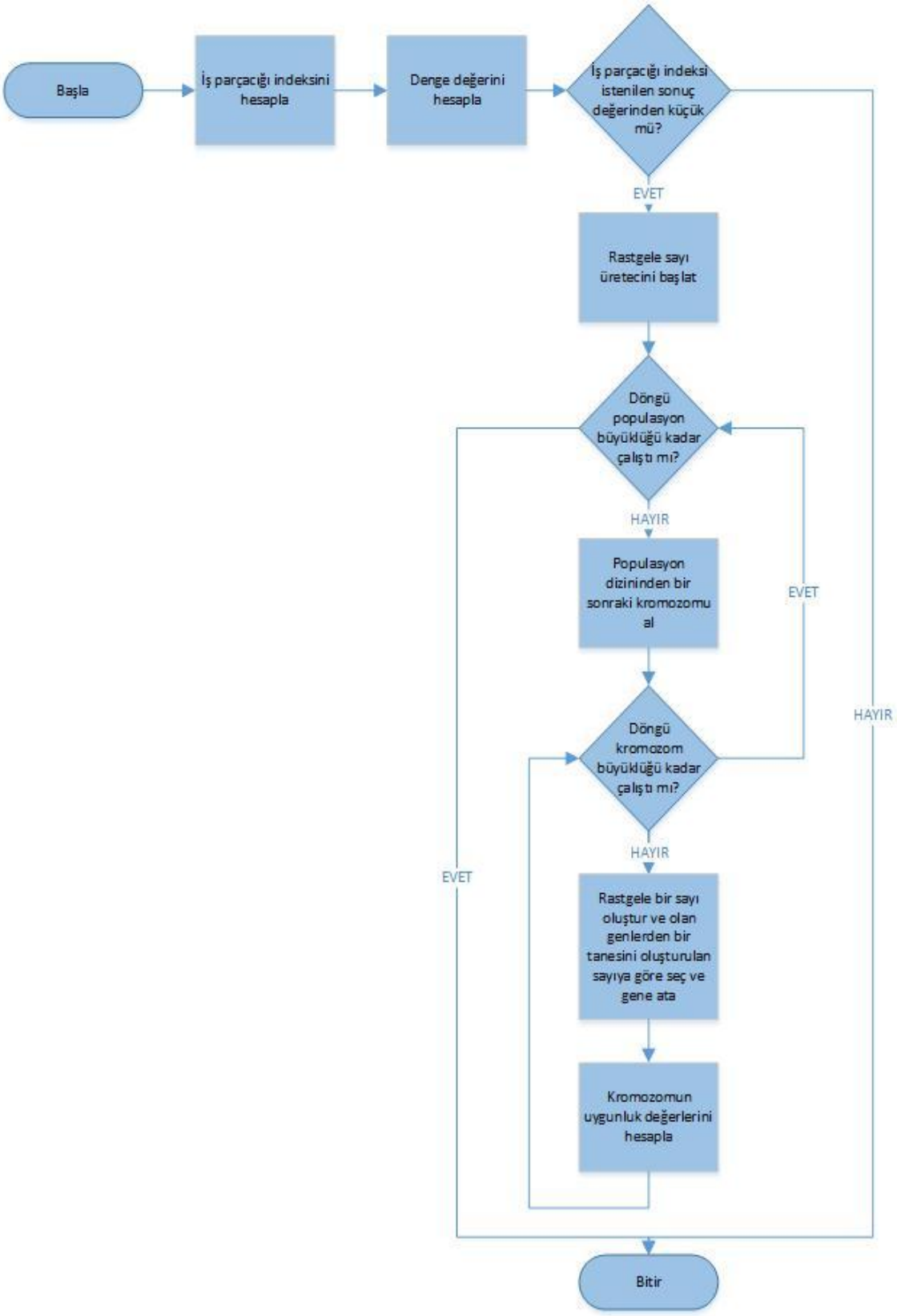


Şekil 4.5 Popülasyon yapısının CPU'dan GPU'ya kopyalanma adımlarının şekilsel gösterimi

Kullanıcı tarafından girilen sonuç sayısı kadar oluşturulan her bir iş parçacığı içerisinde kromozomlara başlangıç değerleri atanmaktadır. Bunun için öncelikle ilgili fonksiyona tüm popülasyonları içeren bir dizi, rastgele sayı üreten fonksiyonun tohum değeri, sonuç sayısı, popülasyon ve kromozomların boyutlarını belirten değerler geçilir. Her bir iş parçacığı, daha önceki bölümlerde belirtilen formülleri kullanarak ızgara ve blok yerleşim düzenindeki yerini hesaplar ve popülasyon dizisi içerisinden kendine ait olan popülasyonu işaret eder. Buradaki önemli nokta, N. iş parçacığının işaret ettiği popülasyon ile N+1. iş parçacığının işaret ettiği popülasyonun başlangıç indeksleri arasındaki farkın bir popülasyon boyutu kadar olmasıdır. Bunun sebebi de GPU üzerindeki popülasyonların CPU üzerindeki gibi 2B dizi ile değil, bütün kromozomların ardışık olarak tutulduğu 1B dizi ile ifade edilmesidir. Doğru popülasyonu işaret etmesi için her bir iş parçacığı, popülasyon büyüklüğü ve paralel yerleşimdeki kendi yerini belirten sayının çarpılması sonucu bir denge (*offset*) değeri hesaplayıp ardından tüm popülasyonların olduğu dizinin başlangıç adresinden bu denge değeri kadar ilerleyerek kendine ait popülasyona erişmektedir. Her iş parçacığı, global hafızada tutulan kendine ait popülasyona eriştikten sonra popülasyondaki her bir kromozomu bir döngü yardımı ile yineleyerek kromozomların her bir genine erişip rastgele bir başlangıç değeri atamaktadır.

Eş zamanlı çalışan her iş parçacığı, genlerin başlangıç değerlerini atamadan önce bir defaya mahsus olarak kendisine geçilen tohum (*seed*) değeri ve o iş parçacığının ızgara ve blok yerleşimi içerisindeki yerini belirten değer ile rastgele sayı üreten fonksiyonu çağırmaktadır. Bu sayede daha önce oluşturulmuş olan rastgele sayı üreten fonksiyondan ardışık olarak rastsal sayı elde edilmekte ve genin başlangıç değeri atanmaktadır. Son olarak başlangıç değerleri atanan her bir kromozom için tüm amaçlara ait uygunluk değerleri hesaplanır ve kromozom yapısında bulunan uygunluk değerleri dizisinde ilgili indekse atanır.

Kromozomların her birine başlangıç değerlerinin atanmasına ait işlemin akış diyagramı şekil 4.6'da ve bu işleme ait sözde kod şekil 4.7'de gösterilmiştir. Akış diyagramındaki işlemler her bir iş parçacığı üzerinde ayrı ayrı ve eş zamanlı olarak yapılmaktadır.



Şekil 4.6 Popülasyondaki her bir kromozoma rastgele ilk genlerin atanmasını gösteren akış diyagramı

```
İstenen sonuç sayısı değerini al;
Tohum değerini al;
İş parçacığı indeksini hesapla;
Denge değerini hesapla;

IF iş parçacığı indeksi < istenen sonuç sayısı değeri
    Rastgele sayı üretecini tohum değeri ile birlikte başlat;
    WHILE popülasyon büyüklüğüne ulaşıncaya kadar
        Popülasyondaki bir sonraki kromozomu al;
        WHILE kromozom büyüklüğüne ulaşıncaya kadar
            Kromozomdaki bir sonraki geni al;
            0 ile 1 arasında rastgele bir sayı üret;
            Gen havuzundan üretilen rastgele sayıya karşılık gelen geni al;
            Gen havuzundan alınan geni kromozomun ilgili genine ata;
        ENDWHILE
        Kromozomun tüm amaçlar için uygunluk değerlerini hesapla;
    ENDWHILE
ENDIF
```

Şekil 4.7 Popülasyondaki her bir kromozoma rastgele ilk genlerin atanmasını gösteren sözde kod

Başlangıç değeri atamaları başarıyla tamamlanan kromozomlar, daha sonra istenilen amaçlara en uygun kromozomların bulunması için GPU üzerinde paralel çalışan genetik algoritmanın koştuğu fonksiyona parametre olarak geçilmektedir. Bu fonksiyona parametre olarak geçilen diğer veriler; en iyi kromozomların listesini tutan bir dizi, genetik algoritma içinde çalışan doğal seçim işlemi için kullanılacak olan diğer kromozom dizisi, sonuç sayısı, popülasyon ve kromozom büyüklükleri, genetik algoritmaya ait parametreler (iterasyon sayısı, çaprazlama ve mutasyon oranı gibi) ve rastgele sayı üreteci için gerekli olan durum dizisidir.

Fonksiyona parametre geçişi yapıldıktan sonra, paralel çalışacak olan fonksiyon daha önce bahsedildiği gibi öncelikle üzerinde koştuğu iş parçacığının ızgara ve blok yerleşimi içindeki lokasyonunu ve global hafızada bulunan tüm veriler içinden kendine ait veriye ulaşabilmesi için kullanılacak olan denge değerini hesaplar. Ardından her bir iş parçacığı üzerinde genetik algoritma kullanılarak bir popülasyon içindeki en iyi kromozomun bulunması işlemi başlar. Algoritmaya ait akış diyagramı şekil 4.8'de gösterildiği gibidir.

Şekil 4.8'de gösterilen akış diyagramında da belirtildiği gibi, öncelikle popülasyondaki kromozomların uygunluk değerlerine göre en iyiden en kötüye göre bir sıralama

yapılmaktadır. Seçici Genetik Algoritma her iterasyon sonrasında konfigürasyon dosyasından okuduğu seçicilik oranı değerine göre belirlediği N sayıda en iyi kromozomu üzerinde hiç bir değişiklik yapmadan bir sonraki iterasyonda kullanılmak üzere sonraki jenerasyona ait kromozom dizisine aktarmaktadır. Bu bir nevi en iyi genetiğe sahip bireyleri korumak ve iterasyon sonunda canlı kalmasını sağlayarak popülasyonun giderek daha iyi bireylere sahip olmasını sağlamaktır. En iyi uygunluk değerine sahip kromozomdan en kötü değerli olana doğru yapılan sıralama sonrasında kromozom dizisinin ilk elemanları en iyileri belirtmektedir.

Şekil 4.8 CUDA iş parçacıkları üzerinde paralel çalışan Genetik Algoritma akış diyagramı

Mevcut jenerasyondan sonraki jenerasyona en iyi kromozomların aktarılması işleminden sonra, birbiri ardına gelecek olan iterasyonlar başlamaktadır. İterasyonların kaç defa yapılacağı bilgisi diğer parametreler gibi genetik algoritmaya ait parametrelerin bulunduğu konfigürasyon dosyasından okunmaktadır. Yeni bir iterasyonda yapılan ilk işlem, genetik algoritmanın üzerinde çalışacağı bir popülasyon oluşturmaktır. Bu popülasyonun bir kısmı önceki iterasyon sonunda elde edilen kromozomların sıralanarak içlerinden belli oranda en iyilerinin aktarılmasıyla oluşturulmaktadır. Popülasyonun geri kalan kısmı ise doğal seçim yöntemi ile oluşturulmaktadır. Doğal seçim yöntemi popülasyon içerisindeki en iyi kromozomları seçmeye meyilli olup aşağıdaki gibi çalışmaktadır:

- Tüm kromozomlara ait uygunluk değerleri toplanıp bir toplam değer elde edilir.
- Popülasyondaki bütün kromozomların uygunluk değerine göre yerleştirildiği rulet tekerleği oluşturulur. Bir kromozomun uygunluk değerinin toplam uygunluk değerine oranı o kromozomun rulet tekerleğinde işgal ettiği alanı belirler. Böylece uygunluk değeri yüksek olan kromozomların uygunluk değeri düşük olanlara göre bir sonraki jenerasyona aktarılma ihtimali arttırılmış olur.
- Popülasyondaki toplam kromozom sayısından, yeni jenerasyona daha önceden aktarılmış olan en iyi kromozomların sayısı çıkarılarak rulet tekerleği yöntemi ile kaç kromozomun bir sonraki jenerasyona aktarılacağı hesaplanır ve hesaplanan sayı kadar bir sonraki adım uygulanır.
- CUDA rastgele sayı üreteci kullanılarak 0 ile 1 arasında rastgele bir sayı üretilir. Üretilen sayının rulet tekerleğindeki hangi alana karşılık geldiği hesaplanır ve o alanın sahibi olan kromozom bir sonraki jenerasyona aktarılır.
- Bir sonraki jenerasyon başarıyla oluşturulduktan sonra yeni iterasyonda artık bu jenerasyon kullanılacağından, yeni jenerasyon mevcut jenerasyonu barındıran diziye kopyalanır.

Rulet tekerleği seçim yöntemi, en iyi kromozomları seçmeye meyilli olduğundan sonraki jenerasyondaki birey çeşitliliğini azaltmaktadır. Bir diğer deyişle, sonraki

popülasyondaki bireylerin en iyi bireylere benzeme ihtimali artmakta ve böylece algoritmanın lokal en iyilere takılma riski ortaya çıkmaktadır. Bu riski azaltmak için de bireyler dengeli bir şekilde birbirlerinden farklılaştırılmalıdır. Sonraki bölümlerde bahsedilecek olan mutasyon işlemi yapılırken bu dengenin bozulmamasına dikkat edilmiş, mutasyon oranı belirlenirken bu gibi kısıtlar göz önüne alınmıştır. Yani, belirlenen mutasyon değeri ile ne çok az sayıda gen mutasyona uğratılarak lokal en iyiye takılma riski arttırılmış ne de gereğinden fazla sayıda gene mutasyon yapılarak akıllı bir algoritma olan seçici Genetik Algoritma için rastgele çalışma durumu ortaya çıkarılmıştır.

Yeni popülasyon yaratıldıktan sonra mevcut iterasyon içinde en iyi kromozomların oluşturulması işlemine geçilmektedir. Tek noktalı çaprazlama ve gen bazlı mutasyon işlemleri bir önceki iterasyondan aktarılan en iyi kromozomlar haricindeki kromozomlara belirli ölçülerde uygulanarak yeni popülasyon içerisindeki en iyi kromozomlar bulunur.

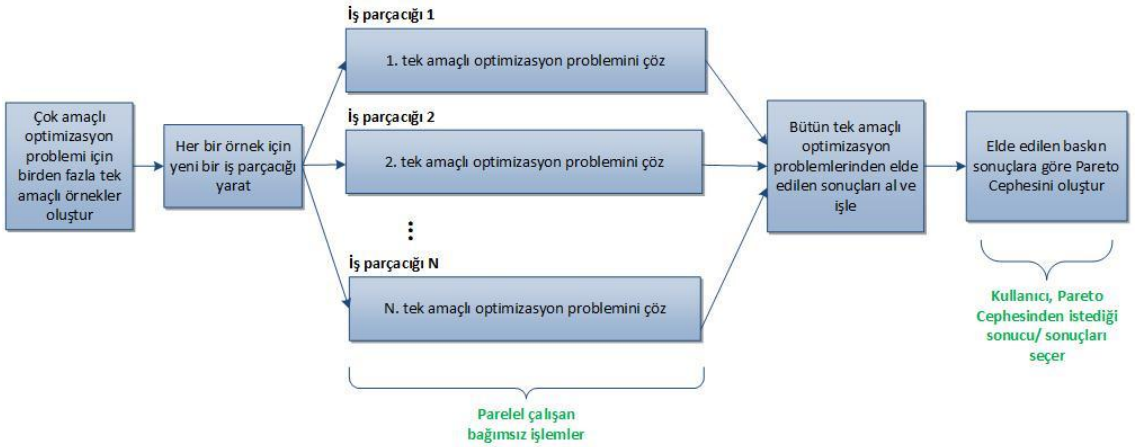
Tek noktalı çaprazlama işlemi, öncelikle kromozom dizisindeki en iyi kromozomlar es geçilerek sonraki kromozomların her ikisi için uygulanmaktadır. Bu işlem için de rastgele sayı üretecinden yararlanılmaktadır. Rastgele sayı üretecinden üretilen 0 ile 1 arasındaki sayı ile daha önce konfigürasyon dosyasından okunan çaprazlama oranı karşılaştırılır; rastgele üretilen sayı çaprazlama oranından küçük ise kromozomlara çaprazlama işlem uygulanır, değil ise o kromozomlar için çaprazlama yapılmaz. Bu işlem için kromozomlar, kromozom dizisinden ardışık olarak seçilmektedir. Çaprazlama işleminin uygulanmasına karar verilen kromozomlar için tekrar bir rastgele sayı üretilmektedir. Üretilen bu rastgele sayı çaprazlamanın hangi genden sonra uygulanacağını belirtmektedir. Örneğin; 16 geni olan kromozomlar için rastgele sayı üretilmiş ve sayı 7 ise, kromozomların 7. ve sonraki genleri çaprazlanacak demektir. Kromozom A ile B'nin genleri 7. gene kadar ve 7. gen ile sonraki genler olmak üzere iki parçaya ayrılır. Kromozom A'nın 7. gene kadarki genleri A1, sonraki genleri A2 olsun. Aynı şekilde kromozom B'nin 7. gene kadarki genleri B1, sonraki genleri B2 olsun. Çaprazlama işlemi sonunda yeni oluşan kromozomlar A1 ile B2 gen dizisinden oluşan A1B2 ve B1 ile A2 gen dizisinden oluşan B1A2 kromozomlarıdır. A1B2 ve

B1A2 kromozomları için yeniden uygunluk değerleri hesaplanır ve bir sonraki kromozom çiftine geçilir. Popülasyondaki tüm kromozom çiftleri için bu süreç aynı şekilde işlemektedir. Çaprazlama işleminin popülasyona etkisi, lokal olarak kromozomların iyileştirilmesine imkan vermesi ve daha iyi kromozomların elde edilmesini sağlamasıdır.

Genetik çeşitliliğin sağlanması ve popülasyonun lokal en iyiye takılmasını önlemek için uygulanan bir sonraki işlem mutasyon işlemidir. Kromozomların her birine gen bazlı mutasyon işlemi uygulanmakta ve yine bunun için de mutasyon oranına göre işlem yapılmaktadır. Gen bazlı mutasyon işlemi de tek noktalı çaprazlama işlemi gibi popülasyondaki en iyi kromozomlara uygulanmamakta ancak diğer kromozomların her biri için aynı süreç işlemektedir. Gen bazlı mutasyon işleminin tek noktalı çaprazlama işleminden uygulama yönünden farkı, kromozomdaki her bir gen için mutasyon işleminden önce mutasyon oranı ile rastgele sayı üretecinden üretilen değer karşılaştırılır; rastgele üretilen sayı mutasyon oranından küçük ise işlem uygulanır, büyük ise uygulanmaz. Bir diğer deyişle, kromozom için değil, her bir gen için mutasyon yapılıp yapılmayacağına karar verilir. Gen bazlı mutasyon işleminin uygulanmasına karar verilen gen için bir rastgele sayı daha üretilir ve o sayıya karşılık gelen muhtemel gen havuzundaki gen, mutasyon uygulanacak olan genin yerine atanır. Gen bazlı mutasyon işlemi tamamlandıktan sonra, oluşan yeni kromozom için uygunluk değerleri hesaplanır. Böylece genetik çeşitliliği sağlanmış yeni kromozomlar ortaya çıkmış olur.

Mevcut iterasyon boyunca uygulanan bir dizi işlemden sonra yaratılan yeni popülasyon içindeki en iyi kromozomların bulunması için popülasyon en iyi uygunluk değerine sahip kromozomdan başlanarak sıralanmaktadır. Sıralanmış olan popülasyon bir sonraki iterasyon sürecine başlanmak üzere yeniden işlenmekte ve bu döngü iterasyon sayısı kadar devam etmektedir. İterasyon sayısı kadar uygulanan döngüde iterasyon sayısına ulaşıldığında mevcut popülasyon artık ulaşılabilen en iyi kromozomu içerisinde barındırmaktadır. Popülasyon içerisindeki en iyi kromozom, GPU üzerinde global hafızada tutulan en iyi kromozomları barındıran dizinin o iş parçacığına ait elemanına atanmaktadır.

Yukarıda bahsedilen genetik işlemler (İterasyon sayısı kadar tekrarlanan doğal seçim, çaprazlama ve mutasyon işlemleri) GPU üzerindeki bir iş parçacığı için anlatılmıştır. Diğer iş parçacıkları içerisinde de aynı işlemler uygulanmaktadır. Böylece GPU üzerinde bütün iş parçacıkları paralel olarak çalıştırılmış ve her bir iş parçacığı içinde ulaşılan en iyi kromozom, en iyi kromozomları barındıran diziye eklenmiş ve GPU'nun görevi bu noktada bitmiştir.



Şekil 4.9 Çekicilik Fonksiyonu temelli Paralel CUDA implementasyonu şekilsel gösterimi

Şekil 4.9'da gösterildiği üzere, çok amaçlı optimizasyon problemi çekicilik fonksiyonu temelli skalarizasyon tekniği ile tek amaca indirgenerek, tek amaçlı optimizasyon problemine dönüştürülmüştür. Tek amaçlı optimizasyon problemi, paralelleştirme yöntemi ile GPU üzerinde atanan yüzler binler mertebesinde iş parçacığı üzerinde eş zamanlı olarak çözülerek ortaya yine yüzlerce/binlerce sonuç çıkarılmıştır. GPU üzerinde elde edilen bu sonuçlar, kullanıcıya sunulmak üzere CPU'ya aktarılmıştır.

4.3 Genel Atama Probleminin Grafik İşlemci Birimi Üzerinde Çözümüne Ait Sonuçlar

Tez çalışması kapsamında Merkezi İşlemci Birimi üzerinde ve Grafik İşlemci Birimi üzerinde çalışan iki farklı uygulama geliştirilmiştir. Merkezi İşlemci Birimi üzerinde çalışan uygulama, test sonuçlarının karşılaştırılması amacı ile Çift Xeon E5-2640 işlemcili ve 16 GB DDR3 RAM özelliklerine sahip HP Z820 iş istasyonunda

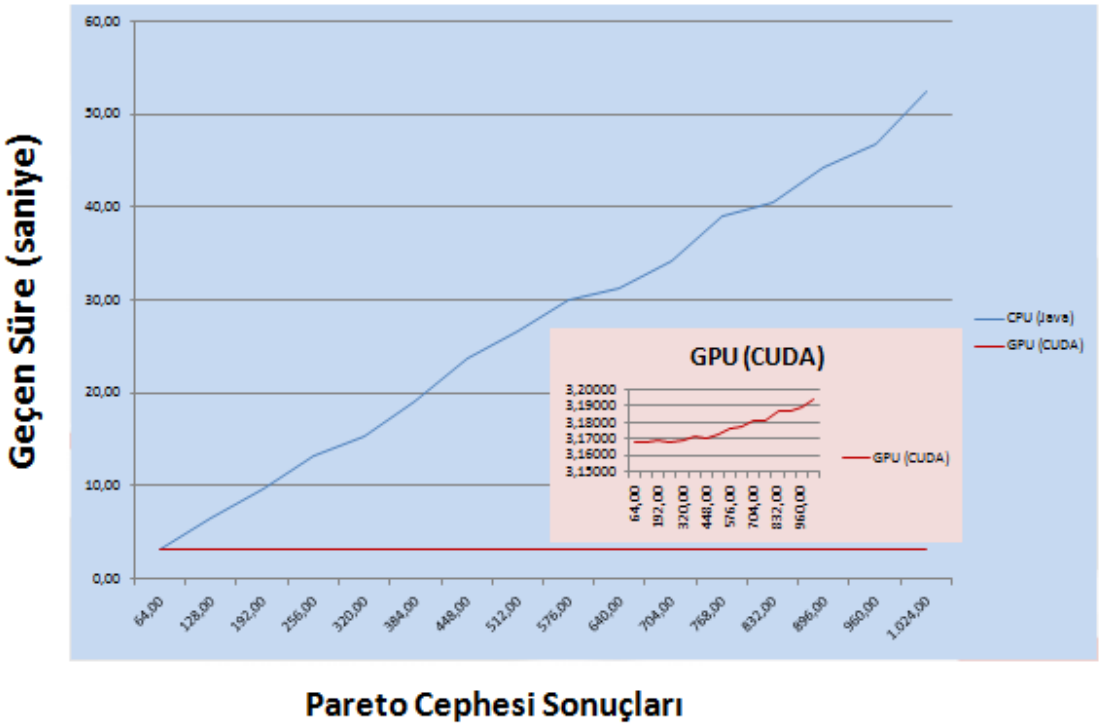
çalışmaktadır. Bu uygulama, JAVA dilinde geliştirilmiş olup kullanıcı tarafından girilen sonuç sayısı kadar çözümü birbiri ardına çalıştırılan algoritma ile bulmaktadır. Test durumları her iki uygulama için aynı olup, problemin çözümü için kullanılan algoritmalar ve diğer parametreler uygulamadan uygulamaya değişmemektir.

Her iki uygulama, bilgisayar üzerinde herhangi bir başka uygulamayı çalışmadığı, işlemci ve grafik kartlarının sadece işletim sistemini ayakta tutmak için minimum seviyede çalıştırıldığı bir ortamda, birbirlerinden farklı zamanlarda test edilmiştir. Böylece bir uygulamayı çalıştırmak için bilgisayar kaynaklarının maksimum seviyede kullanıldığı bir ortam hazırlanmış ve istenen sayıda sonuç elde etmek için uygulamaların çalışma süreleri karşılaştırılmıştır.

Şekil 4.10'da farklı sonuç sayıları için iki uygulamanın harcadığı süreler gösterilmektedir. Grafikte gösterilen uygulamalara ait çalışma süreleri, uygulamaların birden fazla çalıştırılarak elde edilen ortalama çalışma süreleridir. Bunun için Monte Carlo Benzetimi'nden faydalanılmıştır. Her iki uygulama da istenilen her bir sonuç sayısı ile 100'er defa çalıştırılmış, böylece ortalama çalışma süreleri elde edilmiştir.

Şekil 4.10'daki grafikten gözlemlenebildiği üzere, istenen sonuç sayısı çok yüksek değil iken (örn: 10 sonuçtan 200 sonuca kadar) her iki uygulamada da aynı çözümler yaklaşık olarak aynı sürelerde elde edilmektedir. Fakat, grafikte görüldüğü üzere istenen sonuç sayısı arttıkça, merkezi işlemci birimi üzerinde çalışan uygulamanın harcadığı süre doğrusal olarak artmakta, fakat Grafik İşlemci Birimleri üzerinde çalışan uygulamanın harcadığı süre hemen hemen aynı kalmaktadır. Bunun sebebi de önceki bölümlerde bahsedildiği gibi paralelleştirmenin sonuca ve sonuç için harcanan zamana olan olumlu etkisidir. Bir sonraki paragrafta anlatıldığı gibi, her ne kadar paralelleştirme miktarı arttıkça Grafik İşlemci Birimleri üzerinde çalışan uygulamanın çalışma süresi çok az bir artış gösterse de bu artış Merkezi İşlemci birimi üzerinde çalışan uygulamanın çalışma süresindeki artış ile kıyaslanamayacak kadar az miktardadır. Bu yüzden de seri programlama prensibi ile Merkezi İşlemci birimi üzerinde geliştirilen uygulama zaman kritik işlerde kullanıcının isteklerine cevap verse de, işlemlerini makul süre içerisinde tamamlayamamaktadır. Öte yandan Grafik İşlemci Birimleri üzerinde çalışan diğer

uygulama ise hem kullanıcının istekleri doğrultusunda en doğru sonuçları bulmakta, hem de bu işlemi 3 saniye dolaylarında yaptığından oldukça kısa sürede doğru sonuçları bularak kullanıcının süre problemi yaşamasına engel olmaktadır.



Şekil 4.10 Uygulamanın Seri JAVA implementasyonu ile Paralel CUDA implementasyonuna ait sonuçlar

Grafikten gözlemlenebilen bir diğer sonuç ise çok az da olsa grafik işlemci birimi üzerinde çalışan uygulamanın sonuç sayısı arttığında harcadığı zamanın artmasıdır. Bunun sebebi de teoride her ne kadar sonsuz sayıda iş parçacığının eş zamanlı olarak çalışabildiği gösterilse de, pratikte gerek donanım gerekse yazılım kısıtlarından dolayı paralelleştirmenin belli bir oranda kaldığı bu yüzden de sürenin az da olsa arttığı şeklinde açıklanabilir.

Sonuç olarak, her iki uygulamanın harcadığı süreler kıyaslandığında, paralelleştirme oranı arttıkça paralel programlama ile elde edilen zaman tasarrufu açıkça görülmektedir. Bir diğer deyişle, grafik kartları üzerinde çalışan akıllıca geliştirilmiş bir uygulamanın, üzerinde aynı algoritmanın koştuğu merkezi işlemci birimi üzerinde çalışan uygulamadan yaklaşık 16 kata kadar daha hızlı çalışabildiği gösterilmiştir. Çeşitli

yöntemler ile bu farkın daha da arttırılabilmesi mümkün olmakla beraber, bu yöntemlerin bazıları bir sonraki bölümde anlatılacaktır.

5. TARTIŞMA ve SONUÇ

Tez kapsamında Çok Amaçlı Optimizasyon Problemleri'nin Skalarizasyon Teknikleri ile Tek Amaçlı Optimizasyon Problemi haline getirilerek, her bir problemin Grafik İşlemci Birimleri üzerinde eş zamanlı olarak çözülme süreci anlatılmıştır. Bu süreçte ilk olarak Skalarizasyon Teknikleri'nden Ağırlıklı Toplam tekniği ele alınmış ancak bu tekniğin her durumda doğru sonuç vermediği kanıtlanmış, bu yüzden de bir başka Skalarizasyon Tekniği olan Çekicilik Fonksiyonu temelli yaklaşım üzerinde durulmuştur. Çekicilik Fonksiyonu temelli yaklaşımın her durumda her çözüm kümesi elemanları için de doğru sonuç verdiği kanıtlanmış ve uygulamada kullanılmıştır. Daha sonra, GPGPU teknolojisi hakkında bilgi verilmiş, donanıma ve yazılıma ait özellikler anlatılmıştır. GPU'ların CPU'lardan farklı olan çalışma mantığı, mimari yapısı ve GPU üzerinde yazılım geliştirmek için gerekli araç, kütüphane ve geliştirme ortamı ele alınmıştır. Son olarak, daha önce anlatılan iki bölümün tez çalışması kapsamında bir arada kullanılarak bir genel atama probleminin paralel programlama ile nasıl daha hızlı çözülebildiğinden bahsedilmiş, mimari ve tasarım detaylarına girilmiş ve elde edilen sonuçlar paylaşılmıştır. Özet olarak, yapılan çalışma sonunda mevcut CPU altyapısı ile:

- Uzun sürelerde çözülebilen büyük problemlerin GPGPU'lar üzerinde kabul edilebilir (hatta yakın gerçek zamanlı) sürelerde çözülebildiği,
- Çözülemeyecek kadar büyük problemlerin ise GPGPU'lar üzerinde çözülebilir hale getirilebildiği,
- Genelde birleştirme (*aggregation*) teknikleriyle tek amaca indirgenerek çözülen Çok Amaçlı Optimizasyon Problemlerinin, gerçek manada çözümünün oluşturulduğu (yani, Pareto cephesi olarak adlandırılan optimum çözümler kümesinin tamamının sunulduğu)

gösterilmiştir.

Bu bölümde ise, ortaya çıkan uygulamanın geliştirilme sürecinde karşılaşılan problemler ve bu problemlerin çözümüne dair detaylar ele alınmıştır.

Algoritma geliştirme sürecinde karşılaşılan ilk problem hafıza yönetimi ilgili olmuştur. Geleneksel CPU'lar üzerinde geliştirilen yazılım mimarilerinden farklı olarak, GPU'lar üzerinde yazılım geliştirirken hafıza yönetiminin büyük dikkat ile yapılması gerekmektedir. GPU'lar üzerinde hafıza tahsis işlemi tek seferde ve topluca yapılmalıdır çünkü eş zamanlı çalışan her bir iş parçacığı üzerinde ayrı ayrı birden fazla hafıza tahsis işleminin yapılması paralelleştirmeyi bozar. Bu problem ile yazılım tasarım aşamasında karşılaşılmıştır. Çözüm olarak, her bir iş parçacığı üzerinde genetik algoritmanın iterasyon sayısı kadar düzgün bir şekilde çalışabilmesi için gerekli tüm yapıların hafızadaki birim ve toplam boyutları önceden hesaplanmış ve paralel çalışacak olan iş parçacığı sayısı ile çarpılarak tüm yapıların GPU üzerinde işgal ettiği toplam hafıza boyutu bulunmuştur. Daha sonra, toplam büyüklükteki hafıza alanı önceden iş parçacıklarının ve algoritmanın kullanımına tahsis edilmiştir. Böylece her bir iş parçacığı üzerindeki algoritma, ihtiyaç duyduğu yapıyı o esnada yaratıp kullanmaktansa, önceden yaratılmış olan yapılardan kendine ait olanı (bir iş parçacığının global, sabit veya paylaşımlı hafızalar üzerinde bütün veriler arasından kendine ait olanı bulabilmesi için kullanılan yöntem önceki bir önceki bölümde açıklanmıştı) bulup kullanmaktadır.

Bir sonraki problem ile uygulamanın geliştirme aşamasında değil, çalışma esnasında karşılaşılmıştır. Windows işletim sistemi, çevresel cihazlara üzerindeki işlemleri tamamlamaları için varsayılan olarak 2 saniyelik bir süre tanımaktadır. Şekil 4.10'dan da görülebildiği üzere mevcut parametreler ile uygulamanın çalışma süresi yaklaşık 3 saniye civarındadır ve bu süreyi aşmaktadır. Bu sebeple işletim sistemi, GPU üzerindeki uygulamayı işlemini tamamlaması için gereken süre dolmadan 2. saniyede öldürmektedir. Bu problem de Windows kayıt defterindeki ilgili anahtarın değerini yükseltmek ile çözülmüştür. Böylece hem mevcut parametreler ile çalışan hem de parametrelerin uygulamanın çok daha uzun sürelerde çalışmasına sebep olabileceği durumlarda bu problem uygulamanın çalışmasına bir engel teşkil etmez duruma getirilmiştir.

Geliştirme sürecinde tecrübe edilen bir diğer durum ise rastgele sayı üretme süreci ile ilgilidir. Algoritmaya bağlı olarak tamamıyla GPU üzerinde kullanılacak olan rastgele sayıların üretilmesine dair 3 farklı yöntem vardır:

- İlk yöntem sayıların ihtiyaç halinde CPU üzerinde üretilip GPU'ya aktarılmasıdır. Bu yöntem çok etkili olmamakla beraber oldukça yavaş işlemektedir. Ayrıca bu yöntem, paralelleştirme kavramına aykırı olarak çok sayıda CPU-GPU arası veri transferine sebebiyet vermesinden dolayı tercih edilmemiştir.

- İkinci yöntem, sayıların CPU üzerinde tek seferde üretilip daha sonra diğer veriler ile birlikte GPU üzerindeki global hafızaya aktarılması ve ihtiyaç halinde buradan alınıp kullanılmasıdır. Bu yöntem bir öncekine göre daha etkili fakat yine de geliştirilmeye açıktır.

- Son yöntem ise hiçbir CPU-GPU arası veri aktarımına ihtiyaç olmadan, sayıların doğrudan GPU üzerinde ihtiyaç olduğunda üretilmesi ve kullanılmasıdır. Tez çalışmasında diğerlerine göre daha etkili ve hızlı olan bu yöntem tercih edilmiştir.

Çalışma kapsamında, tüm yöntemler test amaçlı olarak denenmiş ve yukarıda bahsedilen durumlar gözlemlenmiştir. Fakat, yapılan çalışmalar ve edinilen tecrübeler doğrultusunda hepsinden daha hızlı çalışabileceği öngörülen yeni bir yöntem kullanılabileceği düşünülmektedir. Bu yönteme dair detaylı bilgi ve uygulamanın performansına doğrudan olumlu etkisi olabilecek diğer geliştirmeler bir sonraki bölümde ele alınacaktır.

6. GELECEK DÖNEMDE YAPILMASI PLANLANAN ÇALIŞMALAR

Geliştirilen uygulamaya dair mimari yaklaşım ile uygulamanın çalışma prensibi "ARAŞTIRMA BULGULARI" bölümünde ele alınmıştı. Ayrıca uygulamanın performansına dair elde edilen sonuçlar da yine daha önce anlatılmış ve Grafik İşlemci Birimleri üzerinde çalışan bu uygulamanın Merkezi İşlemci Birimi üzerinde çalışan uygulamaya oranla çok daha hızlı çalıştığı gösterilmiştir. Son olarak, uygulamanın geliştirilme sürecinde karşılaşılan durumlar çözümleriyle beraber "TARTIŞMA ve SONUÇ" bölümünde paylaşılmıştı. Yazılımın doğası gereği ortaya çıkarılan ürünlerde geliştirme ve iyileştirmeler her daim yapılabilmekte ve hatta ürünün zaman içerisinde yok olmasını engellemek ya da kullanıcının sürekli değişen ve artan ihtiyaçlarına cevap verebilmek adına bu geliştirme ve iyileştirmelerin yapılması zorunlu bir hal almaktadır.

Tez çalışması kapsamında ortaya çıkarılan ürünün performansının da bir takım yazılımsal ve donanımsal geliştirmeler ile daha da arttırılması mümkündür. Bu bölümde ise ürünün performansına olumlu etkisi olacağı düşünülen bir takım yazılımsal ve donanımsal iyileştirmeler anlatılacaktır.

Bir önceki bölümün sonunda rastgele sayı üretme ve kullanma süreci ile ilgili yöntemler anlatılmış ve bu yöntemlerden daha etkin bir yöntem kullanılabileceğinden bahsedilmişti. Bahsedilen yeni yöntem ile sayıların yine GPU üzerinde üretilebileceği ama ihtiyaç halinde değil, tek seferde topyekûn üretilip sabit hafıza veya paylaşımlı hafızada tutulabileceği öngörülmektedir. Böylece hem sayıların üretilmesi anında zamandan tasarruf edileceği hem de tutulduğu hafıza tiplerinin daha hızlı erişim hızlarına sahip olması sebebiyle, iş parçacıkları tarafından bu sayılara erişilirken zaman kaybının daha az olacağı düşünülmektedir.

Mevcut durumda GPU üzerinde çalışan ve CUDA dili ile geliştirilen uygulamanın, CPU üzerinde çalışan ve JAVA dilinde geliştirilen uygulama uygulamaya göre çok daha hızlı sonuçlar ortaya çıkardığı önceki bölümlerde gösterilmişti. Yine de paralel yöntemler kullanılarak geliştirilen algoritmanın bazı düzeltmeler, ek geliştirmeler ve yeniden düzenleme (*refactoring*) teknikleri ile çok daha hızlı çalışabileceği düşünülmektedir.

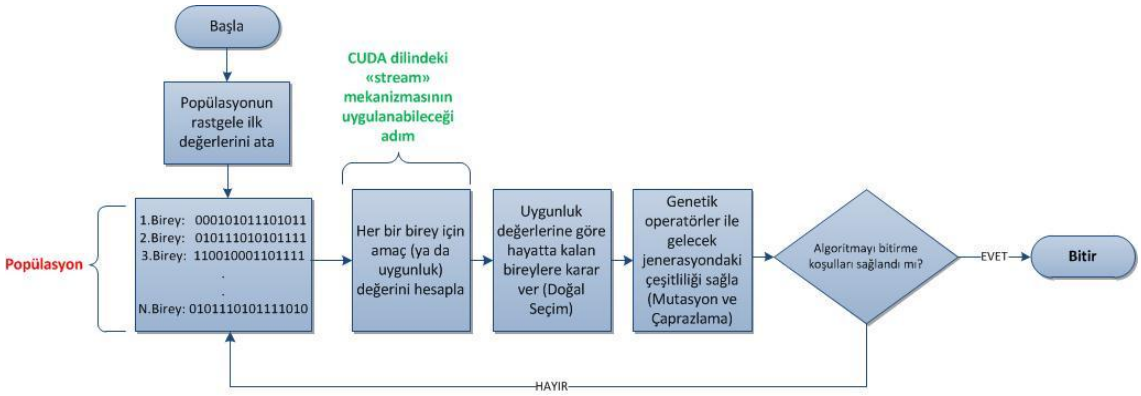
İleride uygulamanın daha performanslı hale getirilebilmesi için yapılabilecek değişiklikler aşağıdaki gibi listelenebilir:

Paylaşımlı hafıza kullanımının arttırılması sağlanabilir. Paylaşımlı hafıza, GPU üzerinde bulunan en hızlı erişilebilirliğe sahip, global hafıza, sabit hafıza ve doku hafızaya göre boyutu daha küçük fakat kullanım açısından daha etkili olan bir hafıza türüdür. Bu hafızanın etkin kullanımı algoritmaya göre, boyutu ise bloklar ve iş parçacıklarının yerleşimlerine göre değişmekle beraber bu çalışmada paylaşımlı hafızaya yer verilmemiştir. Fakat mimari tasarlandıktan ve uygulama geliştirildikten sonra yeniden düzenleme amaçlı bazı değişiklikler ile paylaşımlı hafızanın kullanılabileceği görülmüştür. Örnek olarak; rastgele üretilen sayıların her bir blok için ayrı paylaşımlı hafızada tutularak iş parçacıklarının daha hızlı rastgele işlemleri yapması sağlanabilir.

Algoritmada bulunan dallandırmanın mümkün olduğunca azaltılması sağlanabilir. Algoritmanın doğası gereği yazılım içerisinde sıklıkla rastlanan "if-else" veya "switch" yapıları mevcuttur. Paralel çalışan iş parçacıkları içerisinde var olan bu yapılar, her iş parçacığının birbirinden farklı işler yapmalarına sebep olabilir. Bu da iş parçacıklarının paralel çalışma zamanlarını uzatarak, sonuçta uygulamanın yavaşlamasına sebep olmaktadır. Bu yapıları mümkün olduğunca paralel çalışan kodlar içerisinde kullanmamak, uygulamayı ciddi ölçüde hızlandırmaktadır.

Uygulamanın daha hızlı çalışabilmesi için algoritma içerisinde veya paralelleştirme anında yapılabilecek değişikliklerden biri de CUDA tarafından sağlanan "Stream" mekanizmasını yazılıma dahil etmektir. "Stream" mekanizması uygulamanın paralelleştirme yapıldığı anda uygulanan bir mekanizmadır. Paralelleştirme yapılırken CUDA notasyonları ile iş parçacıklarına ızgara ve blok büyüklükleri ile blok başına düşen paylaşımlı hafıza boyutu gibi çeşitli bilgiler aktarılır. Bu bilgilerin yanında iş parçacıklarına aktarılan diğer bir bilgi ise o iş parçacığının hangi "Stream" ile çalışacağı bilgisidir. CUDA notasyonunda bu alana herhangi bir bilgi yazılmamış ise tüm iş parçacıkları varsayılan "Stream" ile çalışır. Ancak bu alan belli sayıda her iş parçacığı için farklı "Stream" bilgisi ile doldurulursa, o zaman sadece o "Stream" ile çalışan iş

parçacıkları birbiri ile paraleldir ve paralelleştirme seviyesi bir kat daha artmış demektir. Böylece paralelleştirilen iş parçacıkları bir defa daha bölünüp ayrı ayrı çalıştırıldıkları için, kendi içlerinde senkronize olurlar ve diğer paralel çalışan iş parçacığı gruplarından bağımsız işlerini devam ettirebilirler. Tez çalışmasında "Stream" mekanizmasının uygulanabileceği düşünülen algoritma adımı şekil 6.1'de gösterilmiştir.



Şekil 6.1 "Stream" mekanizmasının Genetik Algoritmada uygulanabileceği adım

"Stream" mekanizmasının yazılıma dahil edilmesi, kıyaslama problemleri için çalışan uygulamalarda çok etkili olmayabilir çünkü bu problemler zaten hesaplama bakımından düşük maliyetli test problemleridir. Ancak hesaplama maliyeti oldukça yüksek ve uzun zaman alabilen görev planlama problemi gibi problemlerde diğer tüm değişiklikler gibi böyle bir değişikliğin uygulanmasının etkisi oldukça büyük olacaktır.

Bütün bu bahsedilen değişikliklerin/güncellemelerin gelecek zamanda daha hızlı ve etkili bir grafik kartı olan NVIDIA'nın TESLA model kartları üzerinde yapılması planlanmaktadır. Böylece beklenenden çok daha iyi sonuçların elde edilmesi hedeflenmektedir.

KAYNAKLAR

Akçay, M., Şen, B., Orak, İ., ve Çelik, A. 2011. Paralel Hesaplama ve CUDA. Uluslararası İleri Teknolojiler Sempozyumu (IATS'11). Elazığ Türkiye.

Altınöz, Ö. T., Yılmaz, A. E. and Ciuprina, G. 2013. A Multiobjective Optimization Approach via Systematical Modification of the Desirability Function Shapes. 8th International Symposium on Advanced Topics in Electrical Engineering (ATEE) (s. 1-6). Bucharest: IEEE.

Altıntaş, V. ve Yegenoğlu, E. D. 2011. Görüntü İşlemede Seri ve Paralel Programlamanın Performansı. 6th International Advanced Technologies Symposium (IATS'11).

Anonim. 2009. NVIDIA CUDA Programming Guide Version 2.3.1. NVIDIA Corporation.

Baia, H., OuYang, D., Lia, X., Hea, L. and Yua, H. 2009. MAX-MIN Ant System on GPU with CUDA. Fourth International Conference on Innovative Computing, Information and Control, pp. 801-804.

Bastos-Filho, C. J., Oliveira Junior, M. A., Nascimento, D. N. and Ramos, A. D. 2010. Impact of the Random Number Generator Quality on Particle Swarm Optimization Algorithm Running on Graphic Processor Units. 10th International Conference on Hybrid Intelligent Systems, pp. 85-90.

Bolz, J., Farmer, I., Grinspun, E. and Schroder, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM Transactions on Graphics , 22, 917-924.

Bratton, D., and Kennedy, J. 2007. Defining a Standart for Particle Swarm Optimization. IEEE Swarm Intelligence Symposium, pp. 120-127.

Buck, I. (Tarih yok). High Performance Computing with CUDA. Parallel Programming with CUDA. Web Sitesi: http://mc.stanford.edu/cgi-bin/images/b/ba/M02_2.pdf, Erişim Tarihi: 04.04.2015

Burachik, R. S., Kaya, C. Y., and Rizvi, M. M. 2014. A New Scalarization Technique to Approximate Pareto Fronts of Problems with Disconnected Feasible Sets. Journal of Optimization Theory and Applications , 162 (2), 428-446.

Cardenas-Montes, M., Vega-Rodrigez, M. A., Rodriguez-Vazquez, J. J., and Gomez-Iglesias, A. 2011. Accelerating Particle Swarm Algorithm with GPGPU. 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing(PDP), pp. 560-564.

Castro-Liera, I., Castro-Liera, M. A., and Antonio-Castro, M. C. 2011. Parallel particle swarm optimization using GPGPU. 7th Conference on Computability in Europe (CIE2011).

Catala, A., Jaen, J., and Mocholi, J. A. 2007. Strategies for Accelerating Ant Colony Optimization Algorithms on Graphical Processing Units. IEEE Congress on Evolutionary Computation (CEC 2007), pp. 492-500.

Cecilia, J. M., Garcia, J. M., Nisbet, A., Amos, M., and Ujaldon, M. 2013. Enhancing data parallelism for Ant Colony Optimization on GPUs. Journal of Parallel and Distributed Computing , 73 (1), 42-51.

Cecilia, J. M., Garcia, J. M., Ujaldon, M., Nisbet, A., and Amos, M. 2011. Parallelization Strategies for Ant Colony Optimisation on GPUs. IEEE International Parallel&Distributed Processing Symposium, pp. 339-346.

de P. Veronese, L., and Krohling, R. A. 2009. Swarm's Flight: Accelerationg Particles using CUDA. IEEE Congress on Evolutionary Computation(CEC'09), pp. 3264-3270.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. 2002. A fast and elitist multi-objective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions , 6 (2), 182-197.

Debattisti, S., Marlat, N., Mussi, L., and Cagnoni, S. 2009. Implementation of a Simple Genetic Algorithm within the CUDA Architecture. 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09).

Delevacq, A., Delisle, P., Gravel, M., and Krajecki, M. 2013. Parallel Ant Colony Optimization on Graphics Processing Units. Journal of Parallel and Disributed Computing , 73 (1), 52-61.

Derringer, G., and Suich, R. 1980. Simultaneous optimization of several response variables. Journal of Quality Technology , 12 (4), 214-219.

Fernando, R. 2004. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley Professional.

Fernando, R., and Kilgard, M. J. 2003. The Cg Tutorial. Addison-Wesley Professional.

Fok, K. -L., Wong, T. -T., and Wong, M. -L. 2007. Evolutionary Computing on Consumer-Level Graphics Hardware. IEEE Intelligent Systems , 22 (2), 69-78.

Fu, J., Lei, L., and Zhou, G. 2010. A Parallel Ant Colony Optimization Algorithm with GPU Acceleration Based on All-In-Roulette Selection. Third International Workshop on Advanced Computational Intelligence, pp. 260-264.

Ghorpade, J., Parande, J., Kulkarni, M., and Bawaskar, A. 2012. GPGPU Processing In CUDA Architecture. Advanced Computing: An International Journal(ACIJ) , 3 (1).

Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. 2004. Fast computation of database operations using graphics processors. International Conference on Management of Data, pp. 215-226.

Hillesland, K. E., Molinov, S., and Grzeszczuk, R. 2003. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. ACM Transactions on Graphics , 22, 925-934.

Hung, Y., and Wang, W. 2012. Accelerating parallel particle swarm optimization via GPU. Optimization Methods&Software , 27 (1), 33-51.

Kannan, S., and Ganji, R. 2010. Porting Autodock to CUDA. IEEE World Congress on Computational Intelligence (WCCI 2010), pp. 3815-3822.

Koski, J., and Silvennoinen, R. 1987. Norm methods and partial weighting in multicriterion optimization of structures. International Journal for Numerical Methods in Engineering , 24 (6), 1101-1121.

Krömer, P., Platos, J., Snasel, V., and Abraham, A. 2011. A Comparison of Many-threaded Differential Evolution and Genetic Algorithms on CUDA. Third World Congress on Nature and Biologically Inspired Computing (NaBIC).

Krüger, J., and Westermann, R. 2003. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. ACM Transactions on Graphics , 22, 908-916.

Li, J., Wan, D., Chi, Z., and Hu, X. 2007. An Efficiend Fine-Grained Parallel Particle Swarm Optimization Method Based on GPU-Acceleration. International Journal of Innovative Computing, Information and Control (ICIC International) , 3 (6), 1707-1714.

Marler, R. T., and Arora, J. S. 2010. The weighted sum method for multi-objective optimization: new insights. Structural and Multidisciplinary Optimization , 41 (6), 853-862.

Marler, T., and Arora, J. S. 2005. Function-transformation methods for multi-objective optimization. Engineering Optimization , 37 (6), 551-570.

Mussi, L., and Cagnoni, S. 2009. Particle Swarm Optimization within the CUDA Architecture. 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09).

Mussi, L., Cagnoni, S., and Daolio, F. 2009. GPU-Based Road Sign Detection using Particle Swarm Optimization. Ninth International Conference on Intelligent Systems Design and Applications (ISDA'09), pp. 152-157.

Mussi, L., Daolio, F., and Cagnoni, S. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. Information Sciences , 181, 4642-4657.

Pospichal, P. 2010. GPU-Based Acceleration of the Genetic Algorithm. Pocitacove architektury a diagnostika 2010 (PAD2010), pp. 75-80.

Pospichal, P., Jaros, J., and Schwarz, J. 2010a. Parallel Genetic Algorithm on the CUDA Architecture. Applications of Evolutionary Computation, Volume 6024, pp. 442-451.

Pospichal, P., Schwarz, J., and Jaros, J. 2010b. Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. 16th International Conference on Soft Computing (MENDEL 2010), pp. 64-70.

Saramago, S., and Steffen Jr, V. 1998. Optimization of trajectory planning of robot manipulators taking into account the dynamic of the system. Mechanism and Machine Theory , 33 (7), 883-894.

Schaffer, J. D. 1985. Multiple objective optimization with vector evaluated genetic algorithms. International Conference on Genetic Algorithm and their Applications.

Sel, Ç. 2013. Genel Atama Problemlerinin Çözümünde Deterministik, Olasılık Temelli ve Sezgisel Yöntemlerin Uygulanması. Yüksek Lisans Tezi .

Solomon, S., Thulasiraman, P., and Thulasiram, R. K. 2011. Collaborative Multi-Swarm PSO for Task Matching using Graphics Processing Units. 13Th Annual Conference on Genetic and Evolutionary Computation(GECCO'11).

Srinivas, N., and Deb, K. 1995. Multi-objective function optimization using non-dominated sorting genetic algorithms. Evolutionary Computation , 2, 221-248.

Thompson, C. J., Sahngyun, H., and Oskin, M. 2002. Using modern graphics architectures for general-purpose computing: a framework and analysis. 35th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 306-317.

Trautmann, H., and Mehnen, J. 2009. Preference-based Pareto optimization in certain and noisy environments. Engineering Optimization , 41 (1), 23-38.

Tsutsui, S., and Fujimoto, N. 2009. Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study. 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09).

Tsutsui, S., and Fujimoto, N. 2011. Fast QAP Solving by ACO with 2-opt Local Search on a GPU. IEEE Congress on Evolutionary Computation(CEC), pp. 812-819.

Wong, M., and Wong, T. 2006. Parallel Hybrid Genetic Algorithms on Consumer-Level Graphics Hardware. IEEE Congress on Evolutionary Computation (CEC'06).

Wong, M. -L., and Wong, T. -T. 2009. Implementation of Parallel Genetic Algorithms on Graphics Processing Units. Intelligent and Evolutionary Systems , 187, 197-216.

Wong, M. -L., Wong, T. -T., and Fok, K. -L. 2005. Parallel Evolutionary Algorithms on Graphics Processing Unit. IEEE Congress on Evolutionary Computation (CEC'05), pp. 2286-2293.

You, Y. S. 2009. Parallel Ant System for Traveling Salesman Problem on GPUs. Taiwan Evolutionary Intelligence Laboratory.

Yu, Q., Chen, C., and Pan, Z. 2005. Parallel Genetic Algorithms on Programmable Graphics Hardware. First International Conference on Advanes in Natural Computation(ICNC'05).

Zhou, Y., and Tan, Y. 2009. GPU-based Parallel Particle Swarm Optimization. 11th Annual Conference on Genetic and Evolutionary Computation(GECCO'09).

Zhou, Y., and Tan, Y. 2011. GPU-Based Parallel Multi-Objective Particle Swarm Optimization. International Journal of Artificial Intelligence , 7 (A11).

ÖZGEÇMİŞ

Adı Soyadı : Eren AKÇA

Doğum Yeri : Kütahya

Doğum Tarihi : 27/03/1988

Medeni Hali : Evli

Yabancı Dili : İngilizce

Eğitim Durumu

Lise	:	Dr. Binnaz EGE- Dr. Rıdvan EGE Anadolu Lisesi (2006)
Lisans	:	Çankaya Üniversitesi Mühendislik Fakültesi Elektronik ve Haberleşme Mühendisliği Bölümü (2011)
		Çankaya Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü (Çift Anadal) (2011)
Yüksek Lisans :		Ankara Üniversitesi Fen Bilimleri Enstitüsü Elektrik-Elektronik Mühendisliği Anabilim Dalı (Eylül 2012 - Kasım 2015)

<u>Çalıştığı Kurum ve Yıl</u>

1) HAVELSAN A.Ş (2011 - Günümüz)

Hakemli Dergiler

1) O. T. Altinoz, **Eren Akca**, A. E. Yilmaz,A. Duca, G. Ciuprina, Parallel Implementation of Desirability Function-Based Scalarization Approach for Multiobjective Optimization Problems, Informatica - An International Journal of Computing and Informatics, vol. 39, no. 2, pp. 115-123, 2015. (Compendex)

Uluslararası Kongre Sunum

1) E. Akça., Ö. T. Altınöz., S. U. Emel., A. E. Yılmaz., M. Efe., T. Yaylagül., 2014. Parallel CUDA Implementation of the Desirability-Based Scalarization Approach for Multi-Objective Optimization Problems, International Workshop on Bio-Inspired Optimization Methods and Applications (BIOMA2014) - 13th International Conference on Parallel Problem Solving From Nature (PPSN 2014), pp. 93-105.

Uluslararası Kitap Bölümleri

1) E. Akca, O. T. Altinoz, A. E. Yilmaz, Parallel Implementation of Scalarization Approaches for Multi-objective Optimization Problems, in Advances in Evolutionary Algorithms Research (ed: G. Papa), Nova Publishers, ch. 5, 2015.