T.C.
ERCİYES ÜNİVERSİTESİ
BİLİMSEL ARAŞTIRMA PROJELERİ
KOORDİNASYON BİRİMİ

PARALEL ZEKİ OPTİMİZASYON ALGORİTMALARININ GPU ÜZERİNDE GERÇEKLEŞTİRİLMESİ

Proje No: FBA-11-3520

Normal Araştırma Projesi (NAP)

SONUÇ RAPORU

Proje Yürütücüsü:
Prof. Dr. Adem KALINLI
TIP FAKÜLTESİ/TEMEL TIP BİLİMLERİ

Doç. Dr. Alper BAŞTÜRK
MÜHENDİSLİK FAKÜLTESİ/BİLGİSAYAR MÜHENDİSLİĞİ

Yrd. Doç. Dr. Rüştü AKAY
MÜHENDİSLİK FAKÜLTESİ/MEKATRONİK MÜHENDİSLİĞİ

Aralık 2014
KAYSERİ

TEŞEKKÜR

"Paralel Zeki Optimizasyon Algoritmalarının GPU Üzerinde Gerçekleştirilmesi" konulu araştırma projesi kapsamında kıymetli zamanını ayırarak çalışmalara yaptığı katkılar nedeniyle Birmingham Üniversitesinden Sayın Prof. Dr. Xin YAO'ya ve sağladığı katkılar nedeniyle Üniversitemiz Bilimsel Araştırma Projeleri Koordinasyon Birimi'ne teşekkürlerimizi sunarız.

Prof. Dr. Adem KALINLI

İÇİNDEKİLER

PARALEL ZEKİ OPTİMİZASYON ALGORİTMALARININ GPU ÜZERİNDE GERÇEKLEŞTİRİLMESİ

ÖZET

Büyük boyutlu ve zor problemlerin çözümünde geleneksek yöntemlerin genellikle yetersiz kalması, araştırmacıları yapay zeka optimizasyon algoritmaları gibi yeni ve etkili çözüm yöntemleri geliştirmeye yöneltmiştir. Bu yöntemler geleneksel yöntemlere göre daha başarılı sonuçlar sağlamasına rağmen, büyük boyutlu problemlerin çözümünün genellikle kabul edilebilir sürelerin ötesinde hesaplama zamanı gerektirmesi ayrı bir problem olarak ortaya çıkmaktadır.

Bu problemin aşılması amacıyla da uygulanan yaklaşım genellikle çok işlemcili donanım mimarileri üzerinde paralel hesaplama sistemlerinin kullanılmasıdır. Bununla birlikte geleneksel çok işlemcili paralel mimarilerin kullanımının nispeten zor olması ve yüksek maliyetler gerektirmesi nedeniyle kullanımı çok yaygın olamamaktadır. Grafik işlemci birimlerinin (Graphic Processing Unit, GPU) içerdiği yüzlerce çekirdek ile sahip olduğu doğal paralelliğin genel amaçlı hesaplamalar için kullanılabileceğinin ortaya konması paralel hesaplama konusunda önemli bir dönüm noktası oluşturmuştur. Yakın bir geçmişi olmasına rağmen GPU'ların düşük maliyetleri sebebi ile genel amaçlı problemlerin çözümünde kullanımına yönelik çalışmalar her geçen gün artmaktadır.

Bu kapsamda yapay zeka optimizasyon algoritmalarından bazılarının temel veya geliştirilmiş modelleri GPU üzerinde gerçekleştirilmiştir. Proje çalışmalar kapsamında GPU üzerinde gerçekleştirilen algoritmaların muhtelif test problemleri üzerindeki başarımları incelenmiş ve performans analizleri gerçekleştirilmiştir. Elde edilen sonuçlardan önerilen yapay zeka optimizasyon algoritmalarının grafik işlemci birimleri üzerinde başarıyla kullanılabileceği görülmüştür.

Anahtar Kelimeler: Optimizasyon Algoritmaları, Paralel Programlama, Grafik İşlemci Birimi

1. BÖLÜM
GENEL BİLGİLER

Giriş

Zor problemlerin çözümünde geleneksek yöntemlerin genellikle yetersiz kalması, araştırmacıları yapay zeka optimizasyon algoritmaları gibi yeni ve etkili çözüm yöntemleri geliştirmeye yöneltmiştir. Bu yöntemler geleneksel yöntemlere göre daha başarılı sonuçlar sağlamaktadır. Genel olarak optimizasyon algoritmalarının geliştirilmesi için sosyal, biyoloji, fizik, bilgisayar gibi bilimlerden esinlenilmektedir. Bu tür yaklaşımlara modern sezgisel yaklaşımlar adı verilmektedir. Bu algoritmaların popüler olanlarından bazıları şunlardır: biyoloji biliminin prensiplerini temel alan evrimsel algoritmalar (Evolution Algorithms, EA), zeki problem çözmenin genel kurallarından esinlenerek geliştirilen tabu araştırma algoritması (Tabu Search, TA), karınca kolonilerinin davranışlarını örnek alan karınca kolonisi optimizasyonu (Ant Colony Optimization, KKO), arı kolonilerinin davranışlarını örnek alan yapay arı koloni algoritması (Artificial Bee Colony, YAK), kuş ve balık sürülerini örnek alan parçacık sürüsü optimizasyonu (Particle Swarm Optimization, PSO) ve insan beyninin çalışma prensibinden esinlenilerek geliştirilen yapay sinir ağları (Artificial Neural Network, YSA).

Evrimsel algoritmalar doğadaki evrimsel süreçleri model olarak kullanan bilgisayara dayalı problem çözme teknikleridir. Evrimsel algoritmanın temeli, en iyi uyum sağlayabilenin yaşayabilmesine dayalı biyolojik evrime dayanır. Evrimsel algoritmaların bir alt dalı olan genetik algoritmalarda, doğal seçim ve genetik evrim kurallarına dayanarak geliştirilen bir algoritma modelidir. GA ilk olarak Holland tarafından önerilmiştir [1]. Doğal evrimdeki temel amaç, canlıların yaşadıkları çevrede hayatta kalabilmeleri için gerekli gelişimi sağlamasıdır. GA'nın her çevrimi bir nesil olarak adlandırılır. Başlangıç popülasyonu her nesilde doğal seçme, mutasyon ve çaprazlama gibi genetik operatörlerle geliştirilmeye çalışılır. GA'da genetik işlevler, her bir birey için elde edilen uygunluk değerlerinin değerlendirilmesi ile başlar. Değerlendirme işlevi ile bireylerin uygunluk değerleri baz alınarak bir sonraki nesille aktarılma oranı belirlenir. Ölçeklendirilmiş uygunluk değerleri baz alınarak, bireylerin bir sonraki nesle aktarılma oranı belirlenmiş ve kötü bireyler atılarak yerlerine iyi bireyler yerleştirilmiştir. Böylece çaprazlamaya girecek bireylerin iyi bireylerden oluşması sağlanmış

ve iyi bireylerin sonraki nesille katkıları arttırılmıştır. En son nesil, geçmiş nesillere göre en iyi uyumu sağlayan nesildir [2]. Temel bir GA, paralel yapısı nedeniyle araştırma uzayının etkin bölgelerini oldukça çabuk bulabilir. Ancak olasılık tabanlı yaklaşımlar gösterdiği için birçok durumda bölgesel yakınsama problemine sahip olmakta veya küresel optimuma yakınsaması kabul edilebilir zamanın ötesinde bir süre gerektirebilmektedir. GA araştırmanın geçmiş adımlarına ait bilgileri kümülatif olarak biriktirmekle beraber, değerlendirilmiş çözümlerin bilgisini tutan bir mekanizma kullanmadığı için, benzer yada aynı çözümleri de birçok defa tekrar değerlendirebilmektedir. Bu nedenle GA bazı problemler için aşırı fazla değerlendirme sayısına ihtiyaç duyabilmektedir.

TA algoritması ise zeki problem çözme prensiplerine dayalı olarak Glover tarafından 1986'da önerilmiştir [3,4]. TA algoritması temel olarak belirli bir problem üzerine elde edilen ilk çözüm etrafındaki komşuluklar oluşturmaktır. Komşuluk mekanizması ele alınarak birbirini izleyen seçilmiş çözümlerden daha iyi olanlar hafızaya alınır (Tabu listesi). TA algoritması sahip olduğu bu hafıza ile araştırmanın geçmiş adımları hakkında kayıt tutmakta ve bu kayıtları araştırma uzayında yeni çözümler oluşturulması ve keşfedilmesi için kullanmaktadır. Oluşturulan bu tabu listesi kötü sonuç veren bölgelerde daha fazla işlem yapılmamasını sağlamaktadır. Böylece algoritmanın verimliliği de artmış olmaktadır. Tabu listesinin en önemli özelliklerinden biriside mevcut tabu listesinin aday komşu çözümler ile karşılaştırıldıktan sonra bir sıralama ve karşılaştırma işlemi yaparak kendisini yenileyebilmesidir. TA algoritması iteratif bir algoritmadır ve tek bir çözümden hareketle küresel optimum değere ulaşmaya çalışır. Yasaklama ve serbest bırakma stratejileri ile de algoritmanın bölgesel optimumdan kurtulması sağlanmaya çalışılır. Birçok problem için başarılı sonuçlar üretmesine karşın, TA algoritmasının seri yapısı küresel optimumun bulunduğu bölgeye erişebilmesi için gereken zamanın uzamasına neden olabilmektedir.

KKO, Dorigo ve arkadaşları tarafından önerilen sezgisel algoritmalardan birisidir [5,6]. Algoritma gerçek karıncaların yön ve yiyecek bulma stratejilerine dayalıdır. Karıncalar, yiyecek ile yuvaları arasındaki en kısa yolu bulma kabiliyetine sahiptirler. Karıncalar yürürken yolları üzerine kimyasal bir madde (feromen) bırakmakta ve birbirleriyle haberleşmede bu maddenin yaydığı kokuyu kullanmaktadırlar. Tüm karıncaların hızlarının ve yollara bıraktıkları koku miktarının eşit olduğu kabul edildiğinde, daha kısa yollarda birim zamanda daha fazla koku maddesi oluşacaktır. Dolayısıyla karıncaların büyük çoğunluğu bu en kısa

yolu seçecektir. KKO algoritması yukarda tanımlanan gerçek karınca kolonilerinin yapmış olduğu doğal optimizasyon işleminin yapay bir modelidir. KKO algoritması birçok problemde etkili olmasına karşın giriş veri kümesi büyük olan problemlerde erken yakınsamayla karşı karşıyadır. Bu erken yakınsama problemini aşmak amacıyla KKO algoritması için çeşitli modeller önerilmiştir [7,8,9].

YAK algoritması bal arıları sürüsünün yiyecek arama davranışlarını taklit etmektedir. Algoritmanın ilk modelleri 2005 yılında Karaboğa tarafından önerilmiştir [10, 11]. Yapay arı kolonisi algoritmasında, işçi arılar, gözcü arılar ve kaşif (scout) arılar olmak üzere bir koloni de üç grup arı bulunmaktadır. Algoritma adımlarında her bir çevrim üç adımdan oluşmaktadır: işçi ve gözcü arıların kaynaklara gönderilmesi, gidilen kaynakların nektar miktarlarının hesaplanması ve kaşif arının belirlenerek yeni bir kaynağa rasgele konumlanması. Algoritmada yiyecek kaynakları optimize edilmeye çalışılan problemin olası çözümlerine karşılık gelirken bir kaynağa ait nektar miktarı, o kaynakla ifade edilen çözümün kalite değerini ifade etmektedir. Gözcü arılar rulet tekerleği prensibine göre gidecekleri kaynakları belirlemektedirler. Her kolonide bulunan kaşif arılar ise yiyecek ararken herhangi bir ön bilgi kullanmamakta, tamamen rasgele araştırma yapmaktadırlar. Dolayısıyla arama maliyetleri düşüktür ve de buldukları kaynağın ortalama kalite değeri düşüktür. Buna karşılık kaşif arıların zengin nektar kaynağına sahip keşfedilmemiş kaynakları bulmaları da olasıdır. Özetle YAK algoritmasında gözcü ve işçi arılar keşfedilen kaynaklardan faydalanma işleminde, kaşif arılar ise keşif sürecinde görev alırlar. Gerçek arılarda taşıma hızı koloninin bir kaynağı bulması ve onu kovana getirmesi ile belirlenirken, yapay arılar durumunda bulunan çözümün kalite değeri yani uygunluğu ile belirlenir.

PSO algoritması kuş ve balık sürülerinin sosyal davranışlarını taklit etmektedir. Algoritmanın ilk olarak 1995'te Dr. Eberhart ve Dr. Kennedy tarafından geliştirilmiştir [12]. PSO, Genetik Algoritmalar gibi evrimsel hesaplama teknikleri ile birçok benzerlikler gösterir. Algoritma rastgele çözümlerden oluşan bir popülasyonla başlatılır ve en iyi çözüm bulunana kadar çözümler güncellenerek arama yapılır. PSO'da parçacık denilen potansiyel çözümler, mevcut en iyi çözümleri takip ederek problem uzayında gezinirler. Bu gezinme iki en iyi değere göre gerçekleşir. Bunlardan ilki o ana kadar parçacığın elde ettiği en iyi değer, bir diğeri de o ana kadar popülasyonda tüm parçacıklar tarafından el edilen en iyi çözüm değeridir. Ayrıca

PSO'da çaprazlama ve mutasyon gibi evrimsel operatörler yoktur, gerçekleştirilmesi kolay, ayarlanması gereken parametre sayısı azdır.

Yapay Sinir Ağları, insan beyninin çalışma prensibini kendine model edinmiş yapay sistemlerdir. Bir YSA birbirleri ile paralel ve birbirine bağlı nöronların hiyerarşik organizasyonundan oluşur. YSA'da bilgi geleneksel metotların aksine nöronlar arasındaki bağlantılarda gizlidir. Bu bağlantıların geniş bir alana yayılmış olmaları YSA'ya yayılmış hafıza özelliği verir. Yayılmış hafıza özelliği YSA'ya hata toleransı, eksik bilgi ile çalışabilmek gibi özellikler kazandırır. Bir YSA tabakalar halinde dizilmiş nöronlardan oluşur. Ağ tipine bağlı olarak bir tabakadaki nöron, diğer tabakalardaki nöronlarla veya aynı tabakadaki başka nöronlarla bağlantılıdır. YSA'lar geleneksel metotlara oranla bir problemi, problemin kendi özelliğine ait bir takım matematiksel formüller kullanarak çözmezler, matematiksel formüller yerine problemi örnekler üzerinden öğrenebilirler ve değişen şartlara adapte olabilirler. Bu gibi avantajlarından dolayı özellikle mühendislik de çok geniş bir uygulama alanına sahiptirler [13-14].

Bu proje çalışmasının temel amaçlarından birisi muhtelif optimizasyon algoritmalarının GPU üzerinde gerçekleştirilerek bazı problemlerin çözümünde kullanılmasıdır. Bu amaca yönelik olarak yapılan literatür taraması aşağıda özetlenmektedir

Literatür Taraması

Luong ve ark. (2009) yerel arama yöntemlerini yeni bir metodoloji tasarımı olarak GPU üzerinde gerçekleştirmişler ve önerdikleri yöntemi "permuted perceptron problem" üzerinde test etmişlerdir. Bu çalışmayla GPU kullanımının büyük problem çözümlerini etkili bir şekilde hızlandırdığını ortaya koymuşlardır [15]. Yine aynı yazarlar başka bir çalışmalarında (2010) tabu araştırma algoritması, tepe tırmanma algoritması veya benzeri iteratif yerel arama algoritmalarının GPU üzerinden paralelleştirilebileceğini göstermişler ve bu algoritmalarla genel bir işbirliği modeli oluşturulabileceğini ifade etmişlerdir. Araştırmacılar bu çalışmalarında bir GPU'da bir yerel arama algoritması olmak üzere birden fazla yerel arama algoritması ile oluşturulmuş melez multi-GPU sistemi tasarlamışlardır. Tasarladıkları sistemin CPU üzerinde hesaplamaya göre önemli düzeylerde performans artışı sağladığını göstermişlerdir [16].

Gerardo ve ark. (2009) Parçacık Sürü Optimizasyonu (PSO) algoritmasının literatürdeki bazı paralel modellerini GPU'lar üzerinde gerçekleştirmişlerdir. Geleneksel kişisel bilgisayarlarla GPU'lar üzerinde PSO algoritmasının performansını artırarak yüksek hesaplama gücü elde etmişlerdir [17]. Wojciech ve ark. (2009) tek bir süreç kullanarak çözüm uzayında arama yapan bir meta-sezgisel algoritması için paralel model önermişler ve bu modeli GPU üzerinde gerçekleştirmişlerdir [18]. Yine başka bir çalışmalarında (2010) iş çizelgeleme problemi için GPU üzerinden çalışan çift katlı paralel bir yaklaşım önermişlerdir [19].

Weihang ve ark. 2010 yılında yaptıkları çalışmada tabu araştırma algoritması için bir paralel modeli GPU üzerinde uygulamışlar ve algoritmanın performansını karesel atama problemi (QAP) üzerinde test etmişlerdir. Yapılan çalışmada önerilen model ile geleneksel CPU üzerinde hesaplamaya göre 20 ila 45 kat arası hız artışı sağlandığı elde edilmiştir [20]. Aynı araştırmacılar yaptıkları diğer çalışmalarında ise desen arama algoritması, karınca koloni algoritması ve parçacık sürü optimizasyonu algoritmaları için önerdikleri paralel yapıları GPU üzerinde gerçeklemişler ve gerçekleştirdikleri yaklaşımın performansının CPU üzerinde hesaplamaya göre çok daha yüksek olduğunu göstermişlerdir [20-23].

Wolfgang ve Simon (2009) GPU'ların sayısal uygulamaların hesaplama gücünün önemli bir kaynağı haline geldiğini söyleyerek, modern programlama araçları ile esnek ve rahat bir şekilde GPU üzerinde programlar geliştirilebileceğini göstermişlerdir [24]. Petr Pospichal ve ark. (2010) CUDA yazılım modeli kullanarak genetik algoritmayı GPU bilgisayar platformuna uyarlamışlardır. Uyarlanan modeli Rosenbrock, Griewank ve Michalewicz bencmark fonksiyonlarında performans analizi yapmışlardır. Elde edilen sonuçlar önerilen yaklaşımın problemlerin çözüm kalitesini korurken CPU'lara oranla çok daha hızlı çözüm sağladığını göstermişlerdir. Bu çalışmalarıyla GPU üzerinde hızlandırılmış Genetik Algoritmanın karmaşık ve çözümü zaman alan problemleri kısa sürede çözülebilecek potansiyeli olduğunu ifade etmişlerdir [25].

Sifa ve Zhenming (2009) yaptıkları çalışmada, büyük boyutlu problemlerin çözümü için geleneksel paralel hesaplama ortamının oluşturulmasının çok yüksek maliyetlere gereksinim duyduğuna vurgu yaparak, CUDA yazılım modeli kullanılarak GPU'lar üzerinde gerçekleştirdikleri hiyerarşik paralel genetik algoritmadan bahsetmişler. Oluşturulan bu yapının geleneksel paralel hesaplama ortamlarına oranla çok daha ucuz maliyetlerde yüksek

hızlı çözümler alınabileceğini söylemişlerdir [26]. Lucas ve Renato (2010) Diferansiyel gelişim algoritmasının Grafik işlem birimi (GPU) lar üzerinde C-CUDA kullanarak geliştirdikleri modelin, DE algoritmasının GPU üzerinde bilinen ilk uygulama olduğunu belirtmişler ve modeli 6 farklı test fonksiyonu üzerinde test etmişlerdir. Elde edilen sonuçların çözüm süreleri kıyaslandığında kayda değer bir hız artışı sağlanmıştır [27].

Zor problemler için iyi bir örnek olan gezgin satıcı problemi matematiksel olarak 1930 ların ortalarında formüle edilmiştir. Tanımlaması basit ancak çözümü çok zor olan bu problem Lianming ve ark.(2009) GPU üzerinde gerçekleştirdikleri paralel bağışıklık algoritması ile çözmüşlerdir [28]. Yine benzer bir çalışma Jie ve arkadaşlarının (2010) GPU üzerinde gerçekleştiridikleri paralel karınca koloni algoirtması ile yapılmıştır. Farklı sayılardaki şehirler için gerçekleştirilen gezgin satıcı problemi çözümleri geleneksel paralel hesaplama modellerine göre yapılan çözümlere oranla daha hızlı çözümler elde edilmiştir [29]. GSP problemi için başka bir çalışmada Wang ve arkadaşlarının karınca koloni algoritmasının farklı bir modelini GPU üzerinde gerçekleştirmeleridir. Elde edilen sonuçlarla GPU'nun CPU hesaplamasına göre çok daha verimli olduğu göstermiştir [30]. You ve Ying 2009 yılında yaptıkları çalışmada parçacık sürüsü optimizasyon algoritması için bir paralel modeli GPU üzerinde uygulamışlar ve algoritmanın performansını 4 farklı test fonksiyonunun farklı bilinmeyen sayıları için test etmişlerdir [31].

Luca ve ark. 2009 yılında yaptıkları çalışmada parçacık sürüsü optimizasyon algoritması için bir paralel modeli GPU üzerinde uygulamışlar ve algoritmanın performansını Road Sign Detection problemi üzerinde test etmişlerdir [32]. Aynı araştırmacılar yaptıkları diğer çalışmalarında ise PSO algoritması için NVIDIA'nın farklı ekran kartları ile gerçekleştirdikleri çözümleri ve seri PSO algoritması ile elde ettikleri çözümleri derinlemesine irdelemişlerdir. Elde edilen sonuçları incelendiğinde ise problem boyutunun büyüdükçe hızlanmanın da arttığını söylemişlerdir [33].

Weihang (2010) desen arama algoritması ve diferansiyel gelişim algoritmasını bir arada kullanarak oluşturduğu yeni modeli CUDA ile GPU platformuna uyarlamıştır. Bu model ile gerçekleştirdiği çözümler geleneksel yöntemlere oranla çok daha ucuz maliyetlerde yüksek hızlı çözümler alınabileceğini göstermiştir [34]. GPU üzerinde CUDA kullanarak paralel algoritma geliştirilmesine yönelik yapılan en yeni çalışmalardan birisi Sancı tarafından Nisan

2010 yılında yüksek lisans tezi olarak gerçekleştirilmiştir. Sancı, çalışmasında GSP'nin özelleştirilmiş bir formu olan uçuş rotası planlama probleminin çözümü için bir paralel genetik algoritma modelini GPU üzerinde gerçekleştirmiştir. Yapılan deneysel çalışmalar sonucunda önerilen yaklaşımın geleneksel CPU'ya göre çok daha yüksek performans sağladığı görülmüştür [35].

GPU'ların genel amaçlı problemlerin çözümünde kullanılmasına yönelik çalışmalar son 5-6 yıllık bir geçmişe dayanmasına rağmen, araştırmacılar konuya büyük ilgi göstermişlerdir. Bu kapsamda muhtelif algoritmaların GPU üzerinde gerçeklenmesine ve muhtelif problemlerin çözümüne uygulanmasına yönelik çeşitli sayıda çalışma bulunmasına rağmen henüz üzerinde çalışılması gereken pek çok husus bulunmaktadır. GPU üzerindeki gerçekleştirimlerin performansına yönelik çalışmalar henüz yeterli düzeyde olmadığı gibi, birçok mühendislik problemi üzerindeki başarım çalışmalarının da yapılmasına ihtiyaç duyulmaktadır.

Bu amaçlar doğrultusunda rapor şu şekilde bölümlendirilmiştir:

Bölüm-2'de grafik işlemci birimleri ve programlama modelleri tanıtılmıştır.

Bölüm-3'te optimizasyon algoritmaları hakkında genel bilgiler verilerek gerçekleştirilen optimizasyon algoritmaları tanıtılmıştır. Algoritmalar için gerçekleştirilen performanslarını iyileştirici yaklaşımlar anlatılmıştır.

Bölüm-4'te Algoritmaların basit ve geliştirilmiş modellerinin GPU üzerinde gerçekleştirimlerinin bazı test fonksiyonları üzerindeki hem performansları hem de çalışma zamanları incelenmiştir.

Bölüm-5'te sinyal ve imge işleme problemlerinde yaygın olarak kullanılan hızlı Fourier, ters hızlı Fourier ve radon dönüşümlerinin GPU üzerinde gerçekleştirimleri verilmiştir.

Bölüm-6'de proje çalışmasının kısa bir değerlendirilmesi yapılarak elde edilen sonuçlar yorumlanmıştır.

2. BÖLÜM
GRAFİK İŞLEMCİ BİRİMİ (GPU)

Grafik İşlemci Birimi (GPU) bilgisayarlarda grafik yaratımı için kullanılan aygıtlardır. Modern GPU'lar bilgisayar grafiklerini işleme ve göstermekte son derece verimlidir ve yüksek paralel yapıları nedeniyle de kompleks algoritmalar için oldukça uygundur.

Grafik işlemci birimi ve GPU terimleri bu alanda en büyük üretici olan NVIDIA firması tarafından 1999 yılında isimlendirilmiştir. GPU'lar gelişen teknolojileri ile birlikte yüksek hesaplama gücü sergilemektedirler. Ayrıca geleneksel CPU'ların hesaplama kapasitesi yıllık ortalama 1.4x(piksel/sn) oranında artış gösterirken GPU'lar için bu değer 1.7x-2.3x oranındadır ki buda grafik işlemcilerdeki performans artışının geleneksel işlemcilerin üzerinde olduğunu göstermektedir. Grafik işlemcilerin performansının CPU'lara göre daha fazla artması yapısal farklılıklarından kaynaklanmaktadır. GPU'lar içyapıları, hafıza modelleri ve iş parçacığı gibi kavramlar bakımından da CPU'dan çok farklıdır. İçyapıları bakımından incelendiğinde CPU'larda genellikle asenkron ve bağımsız çalışan 2-8 işlemci çekirdeği bulunurken GPU'lar üzerinde yüzlerce olabilen çok sayıdaki çekirdek bulunur.

GPU üzerinde çalıştırılan yazılımların çoğunlukla C ve C++ programlama dilleri kullanılarak gerçekleştirildiği söylenebilir. C ve C++ programlama dillerine yapılan ekleme sayesinde GPU'ların programlanmasına olanak sağlayan CUDA programlama platformu kolaylıkla kullanılabilir. Ayrıca teknik hesaplamalar ve matematiksel problemlerin çözümü ve analizi için yaygın olarak kullanılan MATLAB yazılımı içinde yeni bazı araç yazılımları ile GPU'lar doğrudan kullanılabilmektedir [37, 38].

CUDA PROGRAMLAMA MODELİ

CUDA, GPU üzerinde oluşturulan paralel bilgi işlem ortamında genel amaçlı matematiksel işlemlerin yapılmasına olanak sağlayan uygulama geliştirme yazılımı ve programlama platformudur [39]. CUDA'nın komut kümesi ve programlama modeli geliştirilirken, yüksek

düzeyli programlama dillerinden C dilinin söz dizimi kullanılarak genişletilmiştir. CUDA ile uygulama geliştirmek için ekran kartlarının bazı donanımsal özellikleri aşağıda verilmiştir.

Ana Sistem – Aygıt (Host-Device)

Uygulama, öncelikle ana sistemde çalıştırılır, daha sonra işlenecek verinin GPU'lara aktarımı gerçekleştirilir. Aynı zamanda CUDA ile yapılan uygulamalarda CPU ve GPU bir arada kullanılabilir. CPU ve GPU'nun bir arada kullanılmasına heterojen programlama denilmektedir. Heterojen programlamada CPU ve GPU'dan ayrı ayrı faydalanarak uygulamalardan yüksek verim almayı amaçlamaktadır.

Koşut işlev (Kernel)

CUDA, uygulama geliştiricilerin, C dili sözdizimini kullanarak yazılım üretmesine olanak sağlamaktadır. Ana sistemde çalıştırılan uygulamanın bir bölümü, koşut işlem yapılması istendiğinde, kernel olarak isimlendirdikleri, özel tanımlanmış bir C işlevi ile aygıta bildirilmektedir. Koşut işlev (kernel) ile, uygulamanın ilgili bölümünün, nasıl bir koşut programlama topolojisi ile çalıştırılacağı programcı tarafından belirtilebilmektedir. "<<< >>>" niteleyicileri kullanılarak, işlevin kaç iş parçacığıyla hangi yapıda çalışacağı tanımlanır.

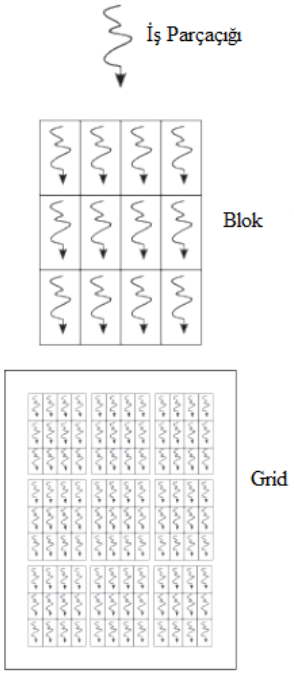
İş parçacığı – Öbek – Izgara (Thread – Block – Grid)

İş parçacığı sıradüzeni, ızgara, blok ve iş parçacığı olarak isimlendirilmiş üç sıra düzensel yapıdan oluşmaktadır. Şekil 1. iş parçacıkları, bloklar ve ızgaralar arasındaki ilişkiyi göstermektedir.

Izgara, iş parçacığı sıradüzeninin en üstünde bulunan yapıdır. Izgara, bloklardan oluşur ve tek ya da iki boyutlu olarak tanımlanabilmektedir. Temel olarak blokların GPU üzerinde yerleşim şeklini tanımlamaya yarar. Izgaranın boyutları, verinin boyutlarına bağlıdır.

Blok, GPU işlemcileri içerisinde çalıştırılacak iş parçacıklarının sayısını ve yerleşim şeklini tanımlamak için kullanılan yapıdır. Tek, iki ya da üç boyutlu olarak tanımlanabilmekte ve en fazla 512 adet iş parçacığı içerebilmektedir.

Koşut programlama için işlevler, thread adı verilen iş parçacıkları tarafından gerçekleştirilmektedir. Bir çekirdek işlevi, tek, iki veya üç boyutlu bir topolojide thread dizileri olarak işlenir.



Şekil 1. GPU içerisindeki İş parçacıkları, Bloklar ve Gridler

Bellek çeşitleri

Ekran kartlarında farklı bellek çeşitleri bulunmaktadır. Bu belleklere erişim hızları birbirinden farklıdır. CUDA, uygulama geliştiricilere bu bellekleri yönetme imkanı verir. Bu bellekler, aşağıdaki özetlenmiştir.

Doku bellek (texture memory) GPU'lar için salt okunur bir bellek türüdür. Belleğe yazma işlemi CPU tarafından gerçekleştirilir, nispeten daha hızlı bellek türdür. 2 veya 3 boyutlu haritalama gibi işlemleri hızlandırmak için kullanılır. Optimizasyon algoritmalarında ise genellikle rasgele sayı üretiminde kullanılır.

Sabit bellek (Constant memory), boyutu dinamik olarak değiştirilemeyen sadece okunabilir bir bellek türüdür.

Bloklar içerisindeki iş parçacıkları arasında iletişim, paylaşımlı bellek (shared memory) ile gerçekleştirilebilir. Paylaşımlı bellek hem okuma hem de yazma erişimi sağlayan çok hızlı bir

bellek türüdür. Ancak sadece blok içinden erişim mümkündür. Tekrarlı erişimler gerektiren veriler için küresel belleğe erişim maliyetini ortadan kaldırır

Yerel bellek ise aslında gerçek bir donanım bileşeni değildir. Derleyici tarafından küresel bellekten tahsis edilmiş bir bellektir.

Küresel bellek (global memory) GPU'nun RAM'idir. CPU ve GPU arasındaki iletişimi sağlar. Bu bellek hem okuma hem de yazma işlemleri sağlayan nispeten yavaş bir bellek türüdür.

İşleme yeteneği (Compute Capability)

Çeşitlilik ve donanımları arasındaki farklılık sebebi ile ekran kartlarının işleme yetenekleri değişmektedir. Dolayısı ile bir aygıtta geliştirilen yazılımın farklı bir aygıt dada kullanılabilmesi için ekran kartlarının genel özelliklerinin aynı olması gerekmektedir.

Programlama Arayüzü

CUDA ile uygulama geliştirmek için C programlama dili uzantısı (C for CUDA) ve CUDA sürücü programlama ara yüzü (CUDA driver API) olmak üzere iki ara yüz sunulmuştur. CUDA programlama modeli, C dili sözdizimini kullanarak yazılım üretilmesine olanak sağlamaktadır. CUDA sürücü programlama ara yüzü ise alt düzey programlama için kullanılmaktadır. Genel olarak uygulamalar bu ara yüzlerden sadece birini tercih ederler ancak teknik olarak ikisin inde aynı program içerisinde kullanılması mümkündür.

Niteleyiciler

CUDA programlama ara yüzü niteleyiciler hakkında kısa bilgiler aşağıda verilmiştir.

İşlev niteleyicileri : İşlev niteleyicileri, işlevin ana bilgisayarda ya da aygıtta çalışacağını belirtmek için kullanılır.

- __device__ : işlevin GPU üzerinde çalışacağını ve sadece GPU'da işlenen yordamlar tarafından çağırılabileceğini belirtir.

- __global__ : işlevin GPU üzerinde çalışacağını ve sadece CPU'da çalışan yordamlar tarafından çağırılabileceğini belirtir.

- __host__ : işlevin CPU üzerinde çalışacağını ve sadece CPU'da çalışan yordamlar tarafından çağırılabileceğini belirtir.

CUDA, özel niteleyici ile tanımlanmayan bütün işlevleri __host__ niteleyicisine sahip olarak görür ve CPU üzerinde çalıştırır [1]. Hem CPU hem de GPU üzerinde çalıştırılmak istenen işlevler __host__ ve __device__ niteleyicilerinin birlikte kullanılması ile geliştirilebilirler.

Değişken niteleyicileri : Değişken niteleyicileri, değişkenlerin aygıt üzerinde hangi çeşit bellek alanında olacağını belirtmek için kullanılır. Bir yordam içinde olmayan ve özel niteleyici ile belirtilmeyen değişkenler bilgisayar belleğinde tutulur.

- __device__ : Değişkenin genel bellekte tutulacağını, uygulama süresince erişilebilir olacağını, bütün iş parçacıkları ile CPU üzerinde çalışan ve CUDA Runtime API kullanan işlevler tarafından erişilebilir olacağını belirtir.

- __shared__ : Değişkenin paylaşımlı bellek üzerinde tutulacağını belirtir. Öbeğin çalışması süresince erişilebilirdir. Aynı öbek içindeki iş parçaları tarafından erişilebilirdir.

- __constant__ : Değişkenin sabit bellek üzerinde tutulacağını, uygulama süresince erişilebilir olacağını, bütün iş parçacıkları ile CPU üzerinde çalışan ve CUDA Runtime API kullanan işlevler tarafından erişilebilir olacağını belirtir.

- volatile : Bellek üzerindeki bir veriye erişimde, okuma işlemi yazma işleminden hızlıdır. Genel veya paylaşımlı bellekte bulunan bir değerin üzerinde bir iş parçacığı tarafından değişiklik yapıldığında, diğer iş parçacıkları tarafından bu değişikliğin görünmesi uygulama içinde genelde mümkün olamamaktadır. Volatile niteleyicisi bunu sağlamak için geliştirilmiştir.

Önceden Tanımlanmış Özel Değişkenler

CUDA'da, aygıt üzerinde çalışan iş parçacıklarının, çalışma anında içinde bulunduğu ızgara ve öbek boyutlarının, öbek ve iş parçacığı değerlerin bilgilerini alabilmeleri için tanımlanmış değişkenler bulunmaktadır.

gridDim: Izgaranın boyut bilgilerini belirtmek için kullanılır. Üç boyutlu bir vektördür. Boyut değerleri tamsayı değerlerdir (gridDim.x, gridDim.y ve gridDim.z). Her boyutta kaç adet öbek olduğu bilgisini tutar.

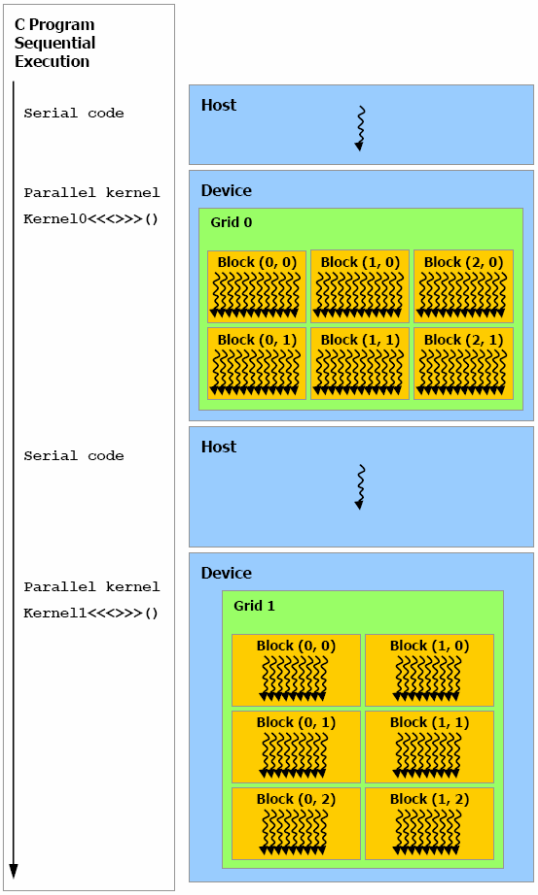
blockDim: Öbeğin hangi boyutunda, kaç adet iş parçacığı bulunduğu bilgisini tutar. gridDim gibi üç boyutlu bir vektördür.

blockIdx: Öbeğin ızgara içindeki yerini gösterir. blockIdx de üç boyut bir vektördür.

threadIdx: İş parçacığının, öbek içindeki yerinin tutan üç boyutlu bir vektördür.

warpSize: Donanımın tek seferde işlediği warp'ın büyüklüğünü belirtmek için tanımlanmıştır. Çok iş parçacıklı uygulamalar için gerek duyulmaktadır.

Şekil 2'de örnek bir GPU uygulaması için program akışı görülmektedir.



Şekil 2. CUDA C için örnek program akışı

GPU donanımları CPU'lardan farklı olduğundan CUDA'da GPU'lar üzerinde program gelişirken kullanılan bazı özel fonksiyonlar bulunmaktadır. Bunlardan en yaygın olarak kullanılanları aşağıda gösterilmiştir. Ayrıca bu özel fonksiyonların kullanıldığı örnek bir CUDA kodu Şekil 3'de gösterilmektedir.

Özel fonksiyonlar

cudaMalloc (void pointer, size_t nbytes):** Doğrusal bellekte alan tahsisi genellikle cudaMalloc fonksiyonu aracılığıyla yapılır.

cudaFree (void* pointer): cudaMalloc ile tahsis edilen alanın temizlenmesi ise cudaFree fonksiyonu aracılığıyla gerçekleştirilir.

cudaMemset (void pointer, int val, size_t nbytes):** Hafızada integer değer ile başlatır.

cudaMemcpy (void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction): Host ve device arasındaki veri transferleri ise cudaMemcpy fonksiyonu ile gerçekleştirilir.

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void KernelSample(int *input,int *output)
{
int x=input[threadIdx.x];
int y=x*x;
output[threadIdx.x]=y;
}
int main()
{
//input array on the host machine
int h_input[6]={2,1,8,6,3,7};
//output array on the host machine
int h_output[6];
//input array on the device machine
int *d_input=0;
//output array on the device machine
int *d_output=0;
//allocation of input array on the device memory
```

```
cudaMalloc((void**)&d_input,6*sizeof(int));
//allocation of output array on the device memory
cudaMalloc((void**)&d_output,6*sizeof(int));
//copy input array from host to device
cudaMemcpy(d_input,h_input,6*sizeof(int),cudaMemcpyHostToDevice);
// invoke kernel for 1 block of 6 threads
KernelSample<<<1,6>>>(d_input,d_output);
//copy output array from device to host
cudaMemcpy(h_output,d_output,6*sizeof(int),cudaMemcpyDeviceToHost);
// free input memory
cudaFree(d_input);
// free output memory
cudaFree(d_output);
return 0; }
```

Şekil 3. Örnek bir CUDA kodu

Matrislerin toplama işlemi için C ve CUDA ile GPU üzerinde gerçekleştirim kodları da aşağıda verilmiştir.

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

Şekil 4. Matris toplamı C kodu

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
int i = threadIdx.x;
int j = threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
}

int main()
{
...
// Kernel invocation with one block of N * N * 1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Şekil 5. Matris toplamı CUDA kodu

3. BÖLÜM
PARÇACIK SÜRÜSÜ OPTİMİZASYON ALGORİTMASI

Temel Parçacık Sürüsü Optimizasyon Algoritması

Parçacık Sürüsü Optimizasyonu (PSO) popülasyon temelli sezgisel bir algoritmadır. Algoritmada problem çözümü, kuşların uzayda yerini bilmedikleri yiyeceği aramalarına benzetilmektedir. Bu amaçla kuş sürülerinin iki boyutlu hareketlerinden esinlenerek Kennedy ve Eberhart tarafından geliştirilmiştir [40]. Algoritmada problem için her bir aday çözüm parçacıkla popülasyon ise sürü ile ifade edilir. Diğer evrimsel algoritmalarda olduğu gibi PSO'da da aday çözümlerden oluşan başlangıç popülasyonu rasgele oluşturulur ve her bir iterasyonda bu çözümler iyileştirilmeye çalışılır. Aday çözümlerin iyileştirilme işlemi ise popülasyonda o andaki en iyi çözüm (gbest) ve kendi kişisel en iyi çözümü (pbest) kullanılarak olur. Bu sayede bireyler arasında bilgi paylaşımı olmaktadır. Algoritma fonksiyon optimizasyonu, bulanık sistemler, yapay sinir ağları gibi birçok alanda basarıyla uygulanabilmektedir [41-46]. PSO algoritmasının temel adımları Algoritma-1 ile gösterilmiştir.

Algoritma 1. PSO algoritmasının temel adımları

```
1: Başlangıç Popülasyonunun Oluşturulması
2: Uygunluk Değerinin Hesaplanması
3: En iyi çözümün belirlenmesi (gbest)
4: Tekrarla
5:     Kendi en iyi değerinin güncellenmesi (pbest)
6:     Hız vektörünün oluşturulması
7:     Yeni aday çözüm üretilmesi
8:     Uygunluk Değerinin Hesaplanması
9:     En iyi çözümün belirlenmesi (gbest)
10: Durdurma Kriteri
```

Popülasyondaki i'ninci çözüm matrisi eşitlik (1) deki gibi, i'ninci çözümün iyileştirilmesi için oluşturulan hız vektörüde eşitlik (2)'deki gibi gösterilebilir.

$$x_i = [x_{i_1}, x_{i_2}, ..., x_{i_D}] \quad (1)$$
$$v_i = [v_{i_1}, v_{i_2}, ..., v_{i_D}] \quad (2)$$

Burada *D* problemin boyutudur. Kendi en iyi değeri ve popülasyondaki en iyi değerin bulunmasından sonra hız vektörü ve yeni komum vektörü aşağıda verilen eşitlik (3) ve (4) ile güncellenir.

$$v_i^{k+1} = v_i^k + c_1.rand_1^k(pbest_i^k - x_i^k) + c_2.rand_2^k(gbest_i^k - x_i^k) \quad (3)$$
$$x_i^{k+1} = x_i^k + v_i^{k+1} \quad (4)$$

Burada, c1 ve c2 öğrenme faktörleridir. c1 ve c2, her parçacığı pbest ve gbest pozisyonlarına doğru çeken, hızlanma terimlerini ifade eden sabitlerdir. c1, parçacığın kendi bilgilerine göre hareket etmesini, c2 ise sürüdeki diğer parçacıkların bilgilerine göre hareket etmesini sağlar. Bu değerlerin düşük seçilmesi araştırma uzayının çok yavaş aranmasına sebep olurken, yüksek seçilmesi, araştırma uzayının hızlı bir şekilde taranması sağlayarak hedefe ulaşmayı hızlandırırken, optimum bölgenin es geçilmesine sebep olabilir. Denklemdeki rand1 ve rand2, [0,1] arasında düzgün dağılımlı rasgele sayılardır. k iterasyon sayısını, i ise parçacık indisini belirtmektedir.

Geliştirilmiş Parçacık Sürüsü Optimizasyon Algoritması

Proje kapsamında PSO algoritmasının performansını iyileştirmek için bazı modifikasyonlar üzerinde çalışmalar yapılmıştır.

Bilindiği üzere PSO algoritmasında yeni üretilen aday çözümler kalitelerine bakılmaksızın mevcut çözümlerle yer değiştirmektedirler. Genel olarak yeni üretilen çözümlerin mevcut çözümün kalitesini koruyacağı ya da iyileştireceği düşünülmektedir. Fakat uygulamalarda her zaman bu şekilde olması mümkün değildir. Yeni üretilen iyileştirilmiş aday çözümler zaman zaman arama uzayında çok farklı bölgelere dağılabilmekte ve iyileşmenin aksine daha da kötüleşebilmektedir.

Önerilen iyileştirilmiş modelde algoritmada n adet kaliteli aday çözümün hiç değişmeden bir sonraki iterasyona taşınması amaçlanmıştır. Bu sayede algoritmada en iyi bireyler muhafaza

edilerek, onların bulunduğu bölgelerin daha fazla araştırılması sağlanacaktır. Bununda algoritmanın genel performansını iyileştireceği düşünülmüştür. Ayrıca popülasyondaki çeşitliğin sağlanması amacı ile aynı amaç fonksiyon değerini üreten aday çözümlerden birinin çıkarılarak yerine rastgele yeni bir aday çözüm üretilmesi sağlanmıştır. İlerleyen bölümlerde iyileştirilmiş PSO algoritması için GPSO kısaltması kullanılacaktır. GPSO algoritmasının temel adımları Algoritma-2 ile gösterilmiştir.

Algoritma 2. GPSO algoritmasının temel adımları

1: Başlangıç Popülasyonunun Oluşturulması
2: Uygunluk Değerinin Hesaplanması
3: En iyi çözümün belirlenmesi (gbest)
4: **Tekrarla**
5: En iyi n adet çözümün hafızaya alınması
6: Hız vektörünün oluşturulması
7: Yeni aday çözüm üretilmesi
8: Uygunluk Değerinin Hesaplanması
9: En kötü n adet çözümün yerine hafızadaki n adet iyi çözümün konulması
10: Çift aday çözümlerin yerine rastgele yeni çözümler üretilmesi
11: Kendi en iyi değerinin güncellenmesi (pbest)
12: En iyi çözümün belirlenmesi (gbest)
13: **Durdurma Kriteri**

4. BÖLÜM
SÜREKLİ TEST FONKSİYONLARININ GPU ÜZERİNDE OPTİMİZASYONU

C ve C-CUDA programlama ile gerçekleştirilen PSO ve GPSO algoritmasının test edilmesi amacıyla CEC2008 yarışması için özelikle geliştirilmiş altı adet sürekli test fonksiyonu incelenmiştir[2]. Tablo-1'de özellikleri ve parametre aralıkları verilen bu fonksiyonlardan her biri farklı karakteristiklere sahiptir. Test problemlerin çözümleri Intel Xeon W3550 @3.07 GHz 8GB RAM, Quadro FX 3800, Windows 7 64-bit özelliklerine sahip bilgisayarda gerçekleştirilmiştir.

Tablo 1. Nümerik test fonksiyonları (U: Tek modlu, M: Çok modlu, S: Ayrışabilen, N: Ayrışamayan).

	İsim	Özellik	Parametre	Parametre Aralığı
1	Sphere	US	100	[-100,100]
2	Schwefel 2.21	UN	100	[-500,500]
3	Rosenbrock	UN	100	[-100,100]
4	Rastrigin	MS	100	[-5.12,5.12]
5	Griewank	MN	100	[-600,600]
6	Ackley	MN	100	[-32,32]

Benzetim çalışmalarında, PSO ve GPSO algoritmalarının C ve C-CUDA ile gerçekleştirimlerinde parametre değerleri tüm test fonksiyonları için aynı alınmıştır. Algoritmalarda popülasyon büyüklüğü 30 alınırken toplam iterasyon sayısı yarışma koşullarını sağlayacak şekilde problem boyutlarına göre ayarlanarak 5000xD alınmıştır [2]. Ayrıca algoritmanın parametrelerinden c1 ve c2 öğrenme faktörleri için literatürde önerilen 1.8, w değeri için 0.6 değerleri alınmıştır. GPSO algoritmasının performansının detaylı analiz yapılabilmesi için test fonksiyonlarının farklı boyutları incelenmiştir.

Tablo-2, 3 ve 4'de nümerik test fonksiyonlarının PSO ve GPSO algoritmaları ile 30 farklı koşma sonucu elde edilen sırasıyla 100, 500 ve 1000 boyutlu test fonksiyonları için ortalama uygunluk ve standart sapma değerleri verilmiştir.

Tablo 2. PSO ve GPSO algoritması ile elde edilen sonuçlar (D=100)

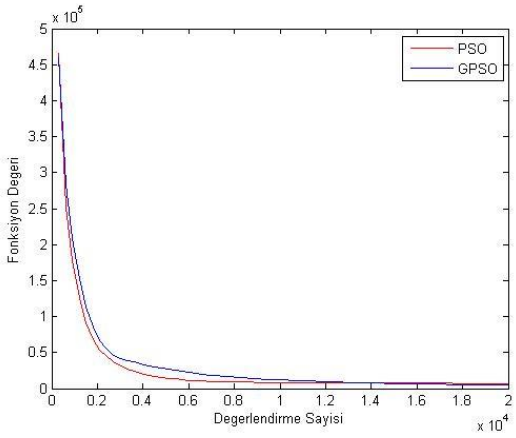
Fonksiyon	PSO		GPSO	
	Ortalama	Standart Sapma	Ortalama	Standart Sapma
f_1	7.17E+03	2.12E+04	**4.08E+00**	1.10E+01
f_2	4.06E+01	7.01E+01	**2.99E+01**	4.62E+01
f_3	2.95E+08	1.73E+09	**4.39E+03**	1.98E+04
f_4	7.99E+02	9.21E+02	**4.72E+00**	9.56E+00
f_5	5.78E+01	1.65E+02	**9.93E-01**	1.10E+00
f_6	1.76E+01	1.94E+01	**1.05E+00**	1.44E+00

Tablo 3. PSO ve GPSO algoritması ile elde edilen sonuçlar (D=500)

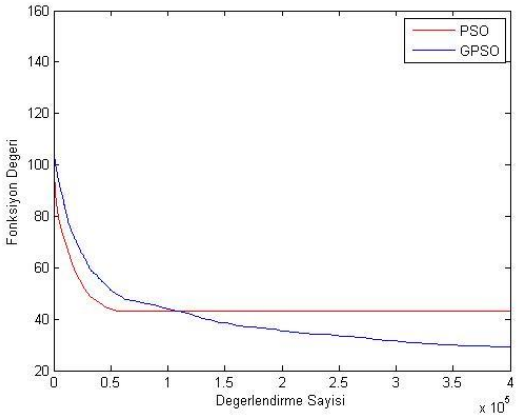
Fonksiyon	PSO		GPSO	
	Ortalama	Standart Sapma	Ortalama	Standart Sapma
f_1	9.57E+04	1.31E+05	**3.49E+01**	4.69E+01
f_2	9.60E+01	1.08E+02	**7.73E+01**	1.02E+02
f_3	8.45E+09	2.30E+10	**2.00E+04**	2.62E+04
f_4	4.39E+03	4.71E+03	**2.29E+01**	3.08E+01
f_5	7.46E+02	1.10E+03	**1.28E+00**	1.52E+00
f_6	1.94E+01	1.95E+01	**1.08E+00**	1.26E+00

Tablo 4. PSO ve GPSO algoritması ile elde edilen sonuçlar (D=1000)

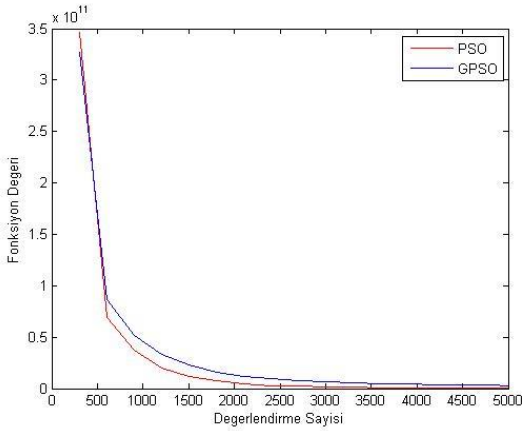
Fonksiyon	PSO		GPSO	
	Ortalama	Standart Sapma	Ortalama	Standart Sapma
f_1	2.58E+05	3.51E+05	**6.53E+01**	8.52E+01
f_2	1.14E+02	1.24E+02	**1.04E+02**	1.24E+02
f_3	3.75E+10	6.69E+10	**3.99E+04**	5.42E+04
f_4	9.27E+02	9.53E+02	**4.62E+02**	5.53E+02
f_5	2.54E+03	3.08E+03	**1.68E+00**	1.81E+00
f_6	1.96E+01	1.97E+01	**1.09E+00**	1.26E+00



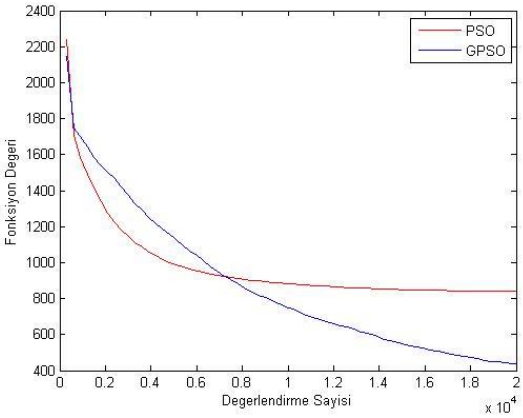
(a) f_1



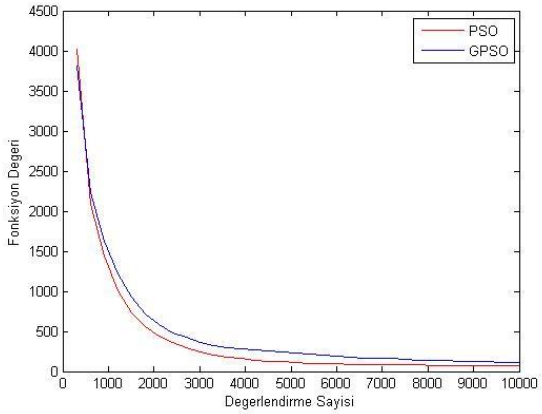
(b) f_2



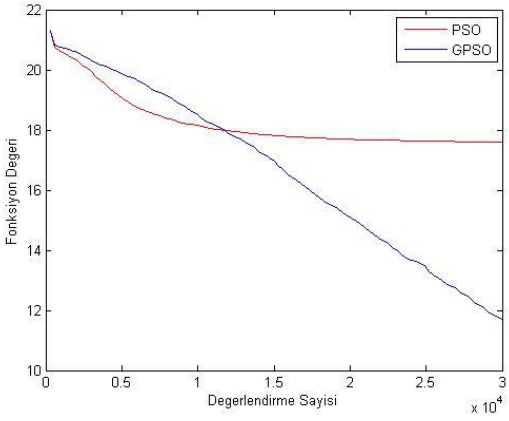
(c) f_3



(d) f_4



(e) f_5



(f) f_6

Şekil 6. PSO ve GPSO algoritmalarının yakınsama grafikleri

Tablo-2, 3 ve 4'de nümerik test fonksiyonlarının PSO ve GPSO algoritmaları ile elde edilen sonuçlar incelendiğinde, GPSO algoritmasının çözüm kalitesinin PSO algoritmasının çözüm kalitesine oranla test fonksiyonlarının tamamında daha iyi olduğu görülmüştür. Yine çok büyük boyutlu test fonksiyonlarının çözümlerinde de GPSO algoritması daha iyi performans göstermektedir. Bunun yanında yakınsama grafikleri incelendiğinde PSO algoritmasının GPSO algoritmasına oranla biraz daha hızlı yakınsadığı söylenebilir. Ancak f_2, f_4 ve f_6 fonksiyonlarının yakınsama grafikleri incelendiğinde PSO algoritmasının hızlı yakınsayarak yerel optimum çözümlere takıldığı görülmektedir. GPSO algoritması ise biraz daha yavaş yakınsama ile PSO algoritmasının takıldığı yerel optimum çözümlere takılmayarak daha kaliteli çözümler üretmektedir.

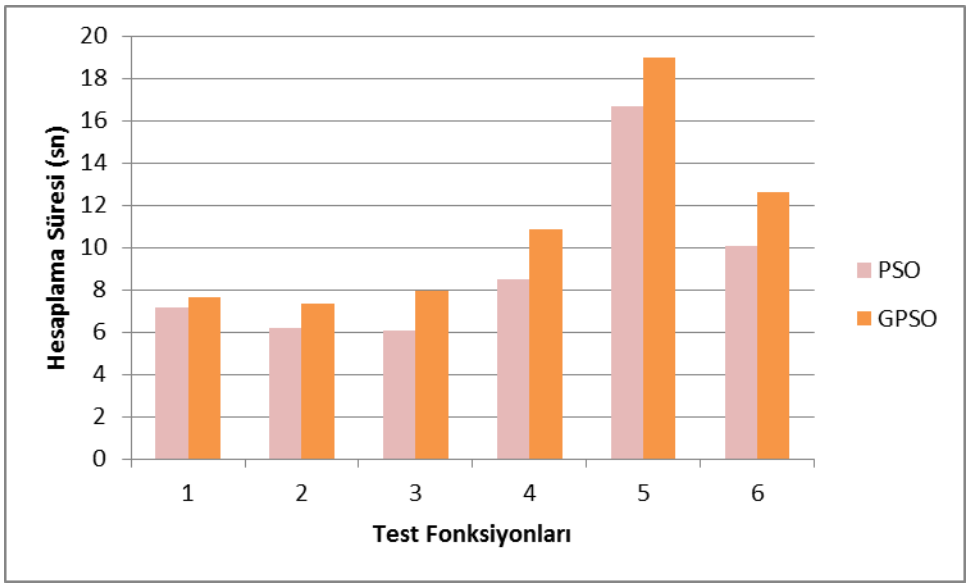
PSO ve GPSO algoritmasının çözüm sürelerinin karşılaştırılması amacı ile C ve C-CUDA programlama ile gerçekleştirim süreleri Tablo-5 ve 6'da sırasıyla 100 ve 500 boyutlu test fonksiyonları için verilmiştir. PSO ve GPSO algoritmasının 100 ve 500 boyutlu test fonksiyonlarının çözüm sürelerinin grafiksel gösterimi sırası ile Şekil 7 ve 8'de verilirken GPSO algoritmasının C ve C-CUDA ile gerçekleşen hesaplama sürelerinin grafiksel gösterimi Şekil 9 ve 10'de verilmiştir.

Tablo 5. PSO ve GPSO algoritmaları için hesaplama süreleri (sn) (D=100)

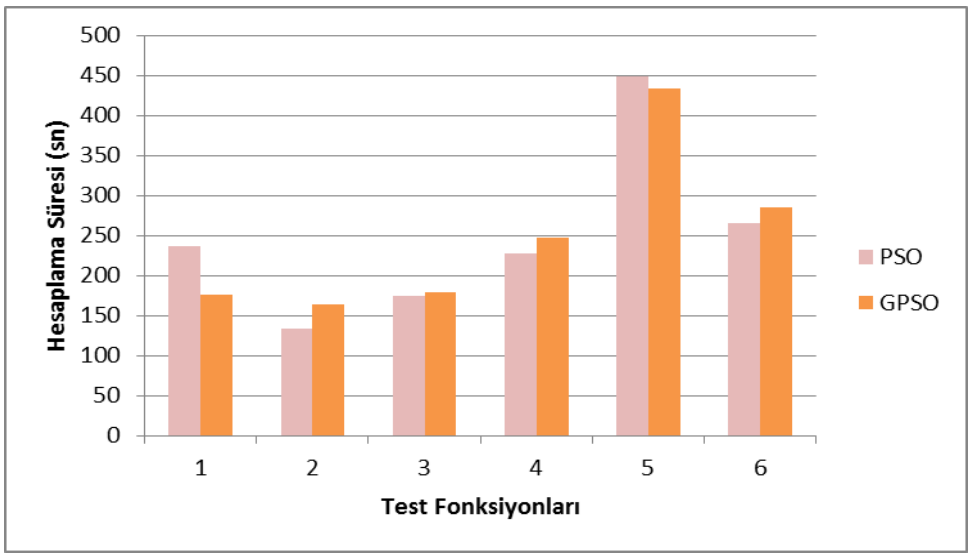
Fonksiyon	PSO		GPSO	
	C	C-CUDA	C	C-CUDA
f_1	7.1660	0.8703	7.6292	1.2950
f_2	6.1616	0.9537	7.3328	1.2430
f_3	6.0564	1.3703	7.9360	1.8650
f_4	8.4684	1.4493	10.8376	1.9370
f_5	16.6552	2.2373	18.9648	3.1513
f_6	10.0552	1.4400	12.5968	1.9817

Tablo 6. PSO ve GPSO algoritmaları için hesaplama süreleri (sn) (D=500)

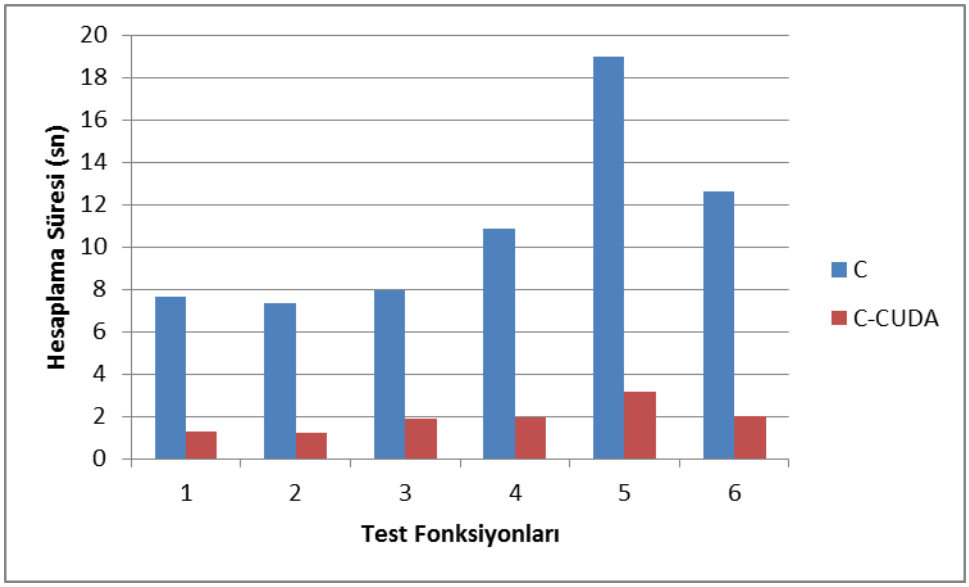
Fonksiyon	PSO		GPSO	
	C	C-CUDA	C	C-CUDA
f_1	237.0560	21.6750	175.1844	16.6680
f_2	132.7524	15.7038	163.0864	18.4047
f_3	175.1168	25.7552	179.3912	26.3532
f_4	227.1528	24.7639	247.4288	26.7884
f_5	449.2980	41.7744	433.5084	40.2790
f_6	265.5592	25.1158	284.3592	26.2479



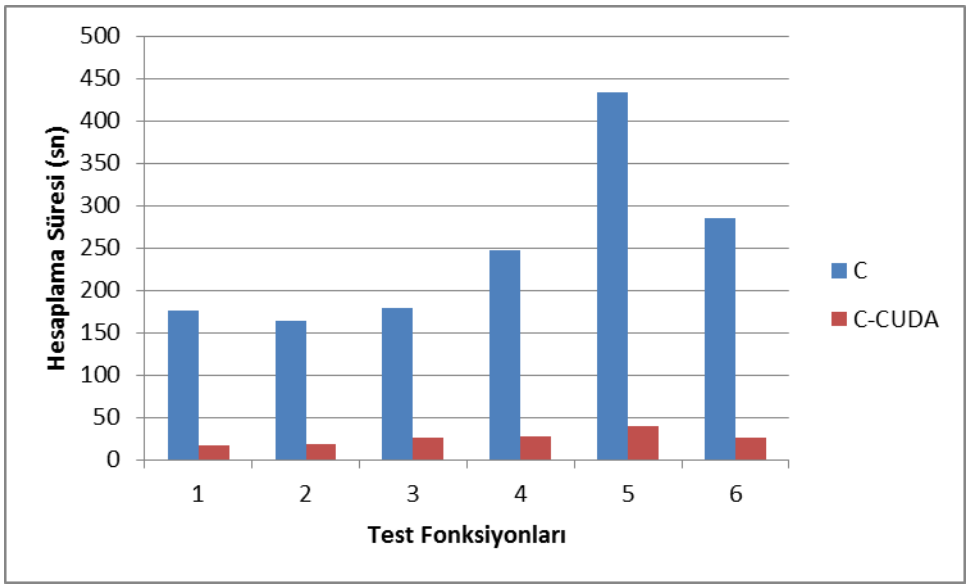
Şekil 7. PSO ve GPSO algoritmasının C hesaplama süreleri (D=100)



Şekil 8. PSO ve GPSO algoritmasının C hesaplama süreleri (D=500)



Şekil 9. GPSO algoritmasının C ve C-CUDA ile hesaplama süreleri (D=100)



Şekil 10. GPSO algoritmasının C ve C-CUDA ile hesaplama süreleri (D=500)

Tablolar ve şekiller incelendiğinde GPSO algoritması ile PSO algoritmasının çalışma zamanları birbirlerine çok yakın çıkmaktadır. PSO algoritmasının iyileştirilmesi amacı ile orijinal kod üzerinde yapılan modifikasyonlar çalışma süresini genel olarak çok az arttırmaktadır.

5. BÖLÜM
İŞARET İŞLEME ALGORİTMALARININ GPU ÜZERİNDE GERÇEKLEŞTİRİLMESİ

Bu bölümde sinyal ve imge işleme problemlerinde yaygın olarak kullanılan hızlı fourier, ters hızlı fourier ve radon dönüşümlerinin GPU üzerinde gerçekleştirimleri yapılmıştır.

Sayısal işaret işleme algoritmalarının gerçek zamanlı olarak GPU üzerinde gerçekleştirilmeye uygun olması bu konuda muhtelif çalışmaların yapılmasına yol açmıştır. Özellikle filtreleme, değiştirme, düzeltme ve sıkıştırma gibi birçok imge işleme tekniğinin temelini teşkil eden ve bazı mühendislik probleminin çözümünde kullanılan Fourier dönüşümü algoritmalarından birisi olan hızlı Fourier dönüşümü (HFD) uygulamaları bu konudaki çalışmaların başlangıcını oluşturmuştur. Yaygın olarak tomografi için kullanılan radon dönüşümü ise görüntü oluşturma olarak bilinir, bu dönüşümle nesneler her açıdan taranarak görüntü oluşturulur.

2003 yılında GPU üzerinde genel amaçlı hesaplama kavramının ortaya konmasından sonra, yüksek performansları ve geleneksel CPU ve paralel işlemcili sistemlerle kıyaslandığında düşük maliyetli olması nedeniyle GPU'ların hesaplama amacıyla kullanılmasına yönelik ilgi artmaya başlamıştır. O yıllarda genel amaçlı algoritmalar Open GL ve DirectX gibi grafik API'leri kullanılarak GPU üzerinde programlanabiliyordu. Ancak, grafik API'leri GPU'nun hesaplama olanaklarından da tam olarak faydalanamıyordu ve bunun sonucu olarak programlama için harcanan eforla kıyaslandığında performans kazancı nispeten düşük kalmaktaydı [47, 52].

HFD'nün GPU üzerinde gerçekleştirilmesine yönelik ilk çalışma 2003 yılında Moreland ve Angel tarafından grafik API'leri ve Open GL programlama ortamı kullanılarak yapılmıştır [47]. Araştırmacılar uygulama neticesinde geleneksel CPU ile kıyaslandığında anlamlı düzeyde bir performans artışı elde edememişlerdir. Ancak, bunun kullandıkları mevcut grafik kartının kapasitesinden kaynaklandığını ve GPU'ların gelişimi ile bu konuda önemli ilerlemeler sağlanabileceğini ifade etmişlerdir [47].

HFD'nün GPU üzerinde gerçekleştirilmesine yönelik literatürde birkaç çalışma bulunmaktadır. Sumamaweera ve Liu tıbbi görüntülerin HDF ile iyileştirilmesi için GPU'nun kullanılmasına yönelik bir yaklaşım önermişlerdir [48]. Ancak onların önerdiği yaklaşım yaklaşık %10 civarında bir performans artışı sağlayabilmiştir. Fielka ve Cadik, NVIDIA GeForce 6600 grafik kartını kullanarak GPU üzerinde görüntü filtrelemede HFD ve konvolüsyon performansını incelemişlerdir. Yaptıkları çalışmada seçilen uygulama üzerinde GPU'nun geleneksel CPU'ya göre 256x256 girişli HFD'nü 3.4 kat daha hızlı gerçekleştirdiğini ortaya koymuşlardır. Genel olarak yapılan çalışmalar NVIDIA firmasının ürettiği GPU'lar üzerine yoğunlaşmıştır. Ancak, Majeed yaptığı yüksek lisans tezinde ilk kez AMD ATI GPU'lar için bir HFD kütüphanesi geliştirmiş ve bir boyutlu ve iki boyutlu HFD'lerini ATI Radeon HD 5870 GPU'su üzerinde gerçekleştirmiştir [47]. Yaptıkları simülasyon çalışmalarında 8192 girişli ve bir boyutlu HFD için Intel Core i7 930 işlemciye göre yaklaşık 3.97 kat; 2048x2048 giriş noktalı ve iki boyutlu HFD için aynı geleneksel CPU'ya göre ise yaklaşık 1.34 kat hız artışı sağlamıştır.

Hızlı Fourier Dönüşümü (HFD)

Fourier dönüşümü esas olarak bir sinyali, bir dalga fonksiyonunu yada herhangi bir matematiksel gösterimi zaman alanından (time domain), frekans alanına (frequency domain) çevirmeye yarar. Fourier dönüşümleri:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft} dt \qquad (5)$$

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{i2\pi ft} df \qquad (6)$$

şeklinde gösterilebilir, burada $-\infty < f < \infty, -\infty < t < \infty,$ ve $i = \sqrt{-1}$ olmak üzere $X(f)$ frekans alanı fonksiyonunu, $x(t)$ zaman alanı fonksiyonunu göstermektedir.

Fourier dönüşümünün sürekli-aperyodik, ayrık-aperyodik, sürekli-peryodik ve ayrık-peryodik, olmak üzere dört tipi mevcuttur. Bilgisayar bilimlerinde genelde ayrık matematik teorisine ihtiyaç duyulduğu için genelde Fourier dönüşümünün ayrık-peryodik olanı kullanılır. Negatif sonsuzdan pozitif sonsuza periyodik bir şekilde kendilerini tekrarlayan ayrık işaretlere ayrık periyodik işaretler denir. Ayrık Fourier dönüşümü (Discrete Fourier Transform, AFD), ayrık periyodik işaretlerin zaman ve frekans alanı dönüşümleri için kullanılır. Ayrık Fourier dönüşümleri:

$$X(k) = \sum_{j=1}^{N} x(j) w_N^{(j-1)(k-1)} \quad (7)$$

$$x(j) = \left(\frac{1}{N}\right) \sum_{k=1}^{N} X(k) w_N^{-(j-1)(k-1)} \quad (8)$$

$$w_N = e^{(2\pi i)/N} \quad (9)$$

şeklinde gösterilebilir, burada $j = 0, 1, \ldots, N;\ k = 0, 1, \ldots, N$ olmak üzere $X(k)$ ve $x(j)$ kompleks serilerdir.

Hızlı fourier dönüşümü (Fast Fourier Transform, HFD), AFD'nün hesaplanması için kullanılan basit ve etkili bir algoritmadır.

HFD algoritmaları, dizinin AFD hesabını daha küçük AFD'lere ayrıştırma temel prensibine dayanmaktadır. Bu temel prensip çeşitli farklı algoritmalarla gerçekleştirilebilmektedir. Çeşitli amaçlara uygun farklı HFD algoritmalarından en sık kullanılanı Cooley-Tukey algoritmasıdır.

Radon Dönüşümü

Literatürde Radon dönüşümü ile ilgili olarak birçok tanım bulunmaktadır. Bunlardan en yaygın olarak kullanılan Radon dönüşümü formu, doğruları $\rho = x\cos(\theta) + y\sin(\theta)$ denklemi aracılığıyla tanımlayan formdur. Burada ρ orijinden doğruya çizilebilecek en kısa doğrunun uzunluğunu ve θ yeni oluşturulan bu doğrunun x-ekseni ile yaptığı açıyı göstermektedir. Bu doğru denklemine bağlı olarak Radon dönüşümü, $g(x, y)$ imgesi boyunca alınan ve ρ ve θ parametrelerine bağlı olarak konumlandırılan bir doğru (hat) integralidir. Böylece Radon dönüşümü

$$\breve{g}(\rho, \theta) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} g(x, y) \delta(\rho - x\cos(\theta) - y\sin(\theta)) dx dy \quad (10)$$

şeklinde verilir [51].

Çalışmada simülasyonlar için NVIDIA Quadro FX 3800 GPU'su üzerinde gerçekleştirilmiş ve sonuçlar Intel'in üst düzey işlemcilerinden ikisi olan 3.07GHz hızındaki Xeon W3550 CPU (CPU1) ve 2.53GHz hızındaki Core 2 Duo P8700 CPU (CPU2)'ları ile elde edilen sonuçlarla karşılaştırılmıştır.

Bir ve iki boyutlu giriş işaretleri için HFD ve ters HFD algoritmaları farklı giriş işareti boyutları için tüm işlemciler üzerinde gerçekleştirilmiştir. Rastgele oluşturulan giriş verileri kullanılarak her bir problem için 100'er deneme yapılmış ve elde edilen sonuçlar Tablo 7-10'da verilmiştir. GPU için verilen süreler GPU'nun hesaplama zamanı ve hafızadaki verilerin PCI veriyolu üzerinden CPU'ya transferi için geçen sürelerin toplamı dikkate alınarak belirlenmiştir.

Tablo 7'den görüldüğü gibi bir boyutlu HFD için küçük boyutlu problemlerde geleneksel CPU'ların daha etkili sonuçlar verdiği görülmektedir. Ancak 256x103 ve daha yüksek boyutlu problemlerde hesaplama süreleri bakımından GPU'nun önemli bir üstünlüğe sahip olduğu görülmektedir. Örneğin 8192x103 için GPU üzerine hesaplama için gerekli sürenin CPU1 ve CPU2 ye göre sırasıyla ile 4.15 kat ve 9.93 kat daha kısa olduğu görülmektedir. Tablo 8'de iki boyutlu problem için elde edilen sonuçlar incelendiğinde ise, GPU'nun bir boyutlu HFD için elde edilenlere oranla daha da yüksek performansa sahip olduğu görülmektedir. Örneğin 2048x2048 boyutlu HFD problemi için GPU'nun CPU1 ve CPU2'ye göre sırasıyla 6.08 kat ve 18.70 kat daha kısa olduğu görülmektedir.

Tablo 7. Bir boyutlu HFD için hesaplama süreleri

İşaret Boyutu ($x10^3$)	GPU (ms)	CPU1 (ms)	CPU2 (ms)
1	0,6864	0,0152	0,0144
4	0,4162	0,0471	0,0347
16	0,5318	0,1437	0,1746
64	0,8274	0,4502	0,5766
256	1,8500	1,6371	3,8678
1024	5,4578	11,1053	26,5887
4096	31,4745	69,4015	153,4490
8192	79,5430	330,4885	790,1409

Tablo 8. İki boyutlu HFD için hesaplama süreleri

İşaret Boyutu	GPU (ms)	CPU1 (ms)	CPU2 (ms)
64x64	0,4050	0,1404	0,8495
128x128	0,2788	0,2754	0,4642
256x256	0,3640	0,9753	1,3469
512x512	0,7653	4,1355	8,1262
1024x1024	2,6772	12,6013	55,3222

2048x2048	12,3925	75,3530	231,7593

Ters HFD uygulamalarında GPU ile hesaplamanın HFD uygulamalarına göre sağladığı performansa oranla anlamlı düzeyde arttığı görülmüştür. Örneğin Tablo 9'den görüldüğü üzere bir boyutlu problemde $8192x10^3$ işaret boyutu için GPU ile CPU1 ve CPU2'ye göre sırasıyla 5.57 kat ve 13.68 kat hız artışı elde edilmiştir. İki boyutlu ters HFD'nün 2048x2048 boyutlu giriş işareti için ise Tablo 10'de görüldüğü gibi CPU1 ve CPU2'ye göre sırasıyla 6.69 kat ve 27.83 kat hız artışı elde edilmiştir.

Tablo 9. Bir boyutlu ters HFD için hesaplama süreleri

İşaret Boyutu ($x10^3$)	GPU (ms)	CPU1 (ms)	CPU2 (ms)
1	0,4891	0,0366	0,0423
4	0,4849	0,0976	0,1075
16	0,5696	0,2931	0,3296
64	0,8615	0,8508	1,1280
256	1,9475	3,8750	5,6460
1024	5,6073	33,2484	45,0132
4096	32,3265	146,2224	280,0955
8192	80,4576	448,8121	1100,5376

Tablo 10. İki boyutlu ters HFD için hesaplama süreleri

İşaret Boyutu	GPU ( )	CPU1 ( )	CPU2 ( )
64x64	0,3552	0,2479	0,1986
128x128	0,2635	0,2773	0,5428
256x256	0,4092	1,1320	2,1284
512x512	0,8205	4,5279	12,2548
1024x1024	2,7797	14,5531	86,2045
2048x2048	12,4538	83,4248	346,5846

Tablo 11. Radon dönüşümü için hesaplama süreleri

İşaret Boyutu	GPU (sn)	CPU1 (sn)	CPU2 (sn)
128x128	0,0792	0,0662	0,0817
256x256	0,3534	0,2646	0,3232
512x512	1,8716	1,0547	1,2910
1024x1024	1,9035	4,2468	5,1972
2048x2048	1,9267	16,2861	20,8490

Bu çalışmada güncel GPU'lardan birisi olan NVIDIA Quadro FX 3800 GPU'su üzerinde bir ve iki boyutlu işaretler için HFD, ters HFD ve RD'leri gerçekleştirilmiş ve özellikle büyük boyutlu problemlerde geleneksel CPU'lara göre hesaplama sürelerinde oldukça önemli düzeylerde azalma sağlandığı görülmüştür.

Grafik işlemciler sahip olduğu yüksek hesaplama gücü ve nispeten düşük maliyetleri ile veri-paralel uygulamalar için yeni ve üstün bir paralel hesaplama ortamı sunmaktadır. Bu güne kadar muhtelif problemlerin çözümünde başarıyla kullanılan GPU'ların Matlab ve C programlama ortamları için geliştirilen kütüphanelerin kullanıma sunulmaya başlamış olmasının sağladığı kullanım kolaylıkları da dikkate alındığında bu etkileyici performansın birçok problemin çözümü amacıyla kullanımına olan ilginin giderek artacağı söylenebilir.

6. BÖLÜM
SONUÇLAR

Büyük boyutlu ve zor problemlerin çözümünde yaygın olarak kullanılan yapay zeka optimizasyon algoritmalarının genellikle kabul edilebilir sürelerin ötesinde hesaplama zamanı gerektirmektedir. Bu çalışmada bu dezavantajın aşılması amacı ile grafik işlemci birimlerinin kullanılması amaçlanmıştır.

Çalışmada ilk olarak, grafik işlemci birimleri hakkında bilgiler edinilerek bazı matematiksel işlemlerin GPU üzerinde gerçekleştirimi yapılmıştır. Gerçekleştirilen bu matematiksel işlemler özellikle büyük boyutlu verilerle yapılan örneklerde geleneksel CPU'lara göre hesaplama sürelerinde önemli düzeylerde azalma sağlandığı görülmüştür. Ayrıca kullanılan ekran kartının özelliklerine göre matematiksel işlemlerin hesaplama sürelerinin de değiştiği gözlemlenmiştir.

Daha sonra yapay zeka optimizasyon algoritmalarından popülasyon temelli sezgisel bir algoritma olan Parçacık Sürüsü Optimizasyonu (PSO) algoritmasının temel modeli CUDA programlama dili kullanılarak GPU üzerinde gerçekleştirilmiştir. Algoritmanın CPU ve GPU çalışma zamanları bazı test fonksiyonları üzerinde incelenmiştir. Ayrıca PSO algoritmasının temel modeli üzerinde bazı modifikasyonlar yapılarak performansını arttırıcı bazı yaklaşımlar gerçekleştirilmiştir. Geliştirilmiş PSO algoritmasının hem performansının hem de çalışma zamanının incelenmesi amacı ile bazı test fonksiyonları kullanılmıştır. Test fonksiyonları çözümlerinin çalışma zamanlarında önemli düzeyde azalma sağlanmıştır. Elde edilen sonuçlardan önerilen algoritmalarının grafik işlemci birimleri üzerinde başarıyla kullanılabileceği görülmüştür.

Ayrıca GPU üzerinde gerçekleştirilen bir ve iki boyutlu işaretler için HFD, ters HFD ve RD'lerinin önemli düzeylerde zamandan kazanç sağladığı görülmüştür.

Gerçekleştirilen optimizasyon algoritmalarının performansının artırılmasına yönelik yeni stratejilerin algoritmaya entegre edilmesi, farklı optimizasyon algoritmalarının GPU üzerinde

gerçekleştirilmesi ve bunların bazı imge ve sinyal işleme problemleri gibi büyük boyutlu mühendislik problemlerinin çözümüne uygulanması, gelecekte yapılabilecek çalışmalar olarak düşünülebilir.

KAYNAKLAR

1. Holland, J.H., Adaption in Natural and Artificial Systems, MAMIT Press, Cambridge, 1975.
2. Goldberg, D.E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, Readin, MA, 1989.
3. Glover, F., Tabu Search – Part I, ORSA Journal on Computing, 1 (3), 190-206, 1989.
4. Glover, F., Tabu Search – Part II, ORSA Journal on Computing, 2 (1), 4-32, 1990.
5. Dorigo, M., Maniezzo, V. and Colorni, A., Positive Feedback as a Search Strategy. Teknik Rapor No. 91-016, Politecnico di Milano, Italy, 1991.
6. Colorni, A., Dorigo, M., Maniezzo, V., Distributed Optimization by Ant Colonies, Proceedings of the first European Conferance on Artificial Life, Paris, France, F.Varela and P.Bourgine (Eds.), Elsevier Publishing, 134-142, 1991.
7. Hirayasu, T., Miki, M., Ono, Y., Ant Colony Optimization for Continous Problems Doshisha University, Dept. of Knowledge Engineering and Computer Science 1-3 Tatara Miyakodani, Kyotanable, Kyoto 610-0321, Japan, 2000.
8. Kalınlı, A., Karaboga, N., Karaboga, D., A Modified Touring Ant Colony Optimisation Algorithm for Continous Functions, The Sixteenth International Symposium on Computer and Information Sciences (ISCISXVI), Işık Universitesi, Antalya, Kasım 5-7, 2001, 94-102, 2001.
9. Sarıkoç F., Paralel Karınca Kolonisi Optimizasyon Algoritması ve Test Problemlerindeki Performansının İncelenmesi, Yüksek Lisans Tezi, Erciyes Üniversitesi, Kayseri, 2004
10. D. Karaboga, An Idea Based On Honey Bee Swarm for Numerical Optimization, Technical Report-TR06, Erciyes University, Engineering Faculty, Computer Engineering Department 2005.
11. D. Karaboga, B. Akay, A Survey: Algorithms Simulating Bee Swarm Intelligence, Artificial Intelligence Review, 31(1), 68-85, 2009.
12. Eberhard, R.C., Kennedy, J., New optimizer using Particle Swarm Theory, Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Nagoya, Japan (1995).

13. Kartalopoulos, S.V., Understanding Neural Networks and Fuzzy Logic, IEEE Pres, New York, 1996.
14. Sagiroglu, S., Artificial Neural Networks in Robotic Applications, International Journal of Mathematical and Computational Applications, 3(2), 67-81, 1998.
15. Luong T. V., et al.:, Local Search Algorithms on Graphics Processing Units. A Case Study: The Permutation Perceptron Problem, Evolutionary Computation in Combinatorial Optimization Lecture Notes in Computer Science, Volume 6022/2010, 264-275, 2010
16. Luong T. V., et al.:, Parallel Local Search on GPU, Institut National De Recherche En Informatiqe Et En Automatique, 2009
17. Laguna-Sanchez G. A., et al., Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU, Journal of Applied Research and Technology, 2009
18. Bozejko W., et al.:, Parallel Calculating of the Goal Function in Metaheuristics Using GPU, Computational Science – ICCS 2009 Lecture Notes in Computer Science, Volume 5544/2009, 2009
19. Bozejko W., et al.:, Parallel hybrid metaheuristics for the flexible job shop problem, Computers and Industrial Engineering archive, Volume 59 , P: 323-333, 2010
20. Zhu W., et al.:, SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration, International Journal of Production Research, Volume 48, p: 1035-1047, 2010
21. Zhu W., Curry J., Parallel ant colony for nonlinear function optimization with graphics hardware acceleration, Proceedings of the 2009 IEEE international conference on Systems, 2009
22. Zhu W., Nonlinear optimization with a massively parallel Evolution Strategy–Pattern Search algorithm on graphics hardware, 2010
23. Zhu W., Curry J., Particle Swarm with Graphics Hardware Acceleration and Local Pattern Search on Bound Constrained Problems, Swarm Intelligence Symposium, 2009
24. Banzhaf W., Harding S., Accelerating evolutionary computation with graphics processing units, Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference , 2009

25. Pospichal P., et al.:, Parallel Genetic Algorithm on the CUDA Architecture, Applications of Evolutionary Computation Lecture Notes in Computer Science, Volume 6024/2010, 2010
26. Zhang S., He Z., Implementation of Parallel Genetic Algorithm Based on CUDA, Advances in Computation and Intelligence Lecture Notes in Computer Science, Volume 5821/2009, 2009.
1. 27. Veronese L.P., Krohling R.A., Differential evolution algorithm on the GPU with C-CUDA, Evolutionary Computation (CEC), IEEE Congress, 2010
27. Li J., Zhang L., Liu L., A Parallel Immune Algorithm Based on Fine-Grained Model with GPU-Acceleration, Fourth International Conference on Innovative Computing, Information and Control, 2009
28. Jie Fu, Lin Lei, Guohua Zhou, A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection, Advanced Computational Intelligence (IWACI), 2010 Third International Workshop, 2010
29. Wang Jiening, Dong Jiankang, Zhang Chunfeng, Implementation of Ant Colony Algorithm Based on GPU, Computer Graphics, Imaging and Visualization, 2009. CGIV '09. Sixth International Conference, 2009
30. You Zhou, Ying Tan, GPU-based parallel particle swarm optimization, Evolutionary Computation, CEC '09. IEEE Congress, 2009
31. Mussi, L., Cagnoni, S., Daolio, F., GPU-Based Road Sign Detection Using Particle Swarm Optimization, Intelligent Systems Design and Applications, 2009
32. Luca Mussia, Fabio Daoliob, Stefano Cagnonia, Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture, Information Sciences, 2010
33. Weihang Zhu, Massively parallel differential evolution-pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems, Springer Science+Business Media, LLC., 2010
34. Sancı S., Parallel Algorithm For Flight Route Planning On GPU Using CUDA. Yüksek Lisans Tezi, ODTÜ Bilgisayar Mühendisliği Bölümü, Nisan 2010
35. A. Basturk, R. Akay, A. Kalınlı, M. E. Yuksel, İşaret İşleme Algoritmalarının GPU üzerinde Gerçekleştirilerek Hesaplama Sürelerinin Azaltılması, 19. Sinyal İşleme ve Uygulamaları Kurultayı SİU-2011, Antalya, 2011
36. http://gp-you.org/download/PDF/GPUmat_User_Modules_0.15.pdf
37. http://www.accelereyes.com/

38. NVidia CUDA, Programming Guide, version 2.3, 2009
39. J. Kennedy and R. Eberhart. Particle swarm optimization. In IEEE International Conference on Neural Networks, 1942–1948, WA, Australia, Nov. 1995.
40. M. Eslami, H. Shareef, M. Khajehzadeh and A. Mohamed, A Survey of the State of the Art in Particle Swarm Optimization, Research journal of Applied Sciences, Engineering and Technology 4(9): 1181-1197, 2012.
41. F. Zhao, Z. Ren, D. Yu, Y. Yang, "Application of An Improved Particle Swarm Optimization Algorithm for Neural Network Training", 0-7803-9422-4/05/2005 IEEE, 2005
42. C. F. Juang, Chao-Hsin Hsu, "Temperature Control by Chip-Implemented Adaptive Recurrent Fuzzy Controller Designed by Evolutionary Algorithm", IEEE, Transactions on Circuits ans Systems-1: Regular Papers, Vol.52, No.11, November, 2005
43. C. F. Juang, C. F. Lu, "Load-frequency control by hybrid evolutionary fuzzy PI controller", IEE Proc.-Gener. Transm. Distrib, Vol. 153, No:2, March 2006
44. H. A. Awad, "A Novel Particle Swarm-Based Fuzzy Control Scheme", IEEE International Conference on Fuzzy Systems, Canada July 16-21, 2006
45. V.R. Pandi and B.K. Panigrahi, Dynamic economic load dispatch using hybrid swarm intelligence based harmony search algorithm. Exp. Syst. Appl., 38: 8509-8851, 2011.
46. Moreland, K., Angel, E., The FFT on a GPU, Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS conference on Graphics hardware session: Simulation and computation, 112- 119., 2003
47. Somanaweera, T., Liu, D., Medical Image Reconstruction with the FFT. In Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley., 2005
48. V.V. Vlcek, Computation of Inverse Radon Transform on Graphics Cards, International Journal of Signal Processing 1(1) ,2004
49. Ryoo, S., Optimization principles and application performance evaluation of a multithreaded gpu using cuda, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming , 2008
50. Peter Toft: "The Radon Transform - Theory and Implementation", Ph.D. thesis. Department of Mathematical Modelling, Technical University of Denmark, June 1996.
51. Majeed N. C. A., Implementation of Fast Fourier Transform on a Graphics Processing Unit, thesis, College of Aeronautical Engineering, September, 2010

52. http://developer.download.nvidia.com/compute/CUFFT_Library_0.8.pdf