T.C.
BALIKESİR ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
ELEKTRİK-ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

PARÇACIK SÜRÜ OPTİMİZASYON ALGORİTMASINDA CUDA KULLANIMININ HIZLANMAYA ETKİSİ

MUHAMMET TAHA AYDIN

YÜKSEK LİSANS TEZİ

Jüri Üyeleri : **Dr. Öğr. Üyesi Gültekin KUVAT (Tez Danışmanı)**
Prof. Dr. Metin DEMİRTAŞ
Doç. Dr. Serdar ÖZYÖN

BALIKESİR, HAZİRAN- 2022

ETİK BEYAN

Balıkesir Üniversitesi Fen Bilimleri Enstitüsü Tez Yazım Kurallarına uygun olarak tarafımca hazırlanan "**Parçacık Sürü Optimizasyon Algoritmasında CUDA Kullanımının Hızlanmaya Etkisi"** başlıklı tezde;

- Tüm bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- Kullanılan veriler ve sonuçlarda herhangi bir değişiklik yapmadığımı,
- Tüm bilgi ve sonuçları bilimsel araştırma ve etik ilkelere uygun şekilde sunduğumu,
- Yararlandığım eserlere atıfta bulunarak kaynak gösterdiğimi,

beyan eder, aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ederim.

Muhammet Taha AYDIN

ÖZET

PARÇACIK SÜRÜ OPTİMİZASYON ALGORİTMASINDA CUDA KULLANIMININ HIZLANMAYA ETKİSİ
YÜKSEK LİSANS TEZİ
MUHAMMET TAHA AYDIN
BALIKESİR ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ
ELEKTRİK-ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI
(TEZ DANIŞMANI: DR. ÖĞR. ÜYESİ GÜLTEKİN KUVAT)
BALIKESİR, HAZİRAN- 2022

Optimizasyon, bir problemin olası tüm çözümleri arasında en iyi çözümün bulunmasıdır. Ancak bazı problemlerin çözümleri kabul edilebilir süreler içerisinde bulunamayabilir. Son yıllarda yaygın olarak kullanılan metasezgisel algoritmalar, problemlerin geçerli bir süre içerisinde iyi bir çözüme ulaşmasını hedeflemektedir. Problemlerin zorlaşması veya boyutlarının büyümesi, başarılı bir çözüm için ihtiyaç duyulan süreyi arttırmaktadır. Başarılı çözümlere daha hızlı ulaşma isteği CUDA'dan faydalanma fikrini ortaya çıkarmıştır. Bu çalışmada parçacık sürü optimizasyon algoritması, CPU, CUDA ve CPU-CUDA hibrit yapıda olmak üzere üç farklı şekilde uygulanmıştır. CPU'da seri programlama, CUDA ve CPU-CUDA hibrit yöntemde paralel programlama uygulanarak 64 ve 128 boyutlu 18 farklı test fonksiyonu çözülmüştür. Farklı iterasyon sayıları için elde edilen en iyi, en kötü, ortalama, standart sapma sonuçları ve çalışma süreleri verilmiştir. Her bir durum için CUDA ve CPU-CUDA yöntemlerinin hızlanma değerleri hesaplanmıştır. CUDA yönteminde en yüksek hızlanma 9,120834 ve en düşük ise 1,927629 olarak bulunmuştur. CPU-CUDA hibrit yöntemde ise en yüksek 7,136033 ve en düşük 1,046644 hızlanma değeri elde edilmiştir. Ortalama hızlanma sonuçlarına göre CUDA, CPU-CUDA hibrit yönteme göre problem boyutu 64 olduğunda yaklaşık 2 kat, 128 olduğunda ise yaklaşık 1,5 kat daha hızlı çalışmıştır.

ANAHTAR KELİMELER: CUDA, Paralel Programlama, Parçacık Sürü Optimizasyonu

Bilim Kod / Kodları: 90542 Sayfa Sayısı: 59

ABSTRACT

THE EFFECT OF CUDA USAGE ON SPEEDUP IN PARTICLE SWARM OPTIMIZATION ALGORITHM
MSC THESIS
MUHAMMET TAHA AYDIN
BALIKESIR UNIVERSITY INSTITUTE OF SCIENCE
ELECTRICAL AND ELECTRONICS ENGINEERING
(SUPERVISOR: ASSIST. PROF. DR. GÜLTEKİN KUVAT)
BALIKESİR, JUNE - 2022

Optimization is the finding of the best solution among all possible solutions to a problem. However, the solutions to some problems may not be found within acceptable time limits. Metaheuristic algorithms, which have been widely used in recent years, aim to reach good solutions for the problems within a reasonable time. The difficulty or size of the problems increases the time needed for a successful solution. The desire to reach successful solutions faster has led to the idea of utilizing CUDA. In this study, the particle swarm optimization algorithm is applied in three different ways: CPU, CUDA and CPU-CUDA hybrid structure. By applying serial programming in CPU, parallel programming in CUDA and CPU-CUDA hybrid method, 18 different test functions with 64 and 128 dimensions are solved. The best, worst, mean, standard deviation results and run times, which we obtain, are given for different iteration numbers. The speedup values of CUDA and CPU-CUDA methods are calculated for each case. In the CUDA method, the highest and lowest speedup values are found as 9.120834 and 1.927629, respectively. In the CPU-CUDA hybrid method, those values are obtained as 7,136033 and 1,046644, respectively. Comparing CUDA with the CPU-CUDA hybrid method in terms of their average speedup results, CUDA runs approximately 2 times faster when the problem size is 64 and 1.5 times faster when it is 128.

KEYWORDS: CUDA, Parallel Programming, Particle Swarm Optimization

Science Code / Codes: 90542 Page Number: 59

İÇİNDEKİLER

ÖZET .... **i**
ABSTRACT .... **ii**
ŞEKİL LİSTESİ .... **iv**
TABLO LİSTESİ .... **v**
KISALTMALAR LİSTESİ .... **vi**
ÖNSÖZ .... **vii**
1. GİRİŞ **1**
2. PARALEL HESAPLAMA **3**
2.1 Von Neumann Bilgisayar Mimarisi 5
2.2 Hızlanma Faktörü 7
2.3 Paralel Bilgisayar Bellek Mimarisi 9
2.4 Paralel Programlama 10
2.5 CUDA Mimarisi ve Programlaması 13
2.5.1 CUDA Donanım Mimarisi 13
2.5.2 CUDA Yazılım Mimarisi 15
2.6 CUDA Uygulama Alanları 20
2.6.1 CUDA Derin Sinir Ağı (CUDA Deep Neural Network-cuDNN) 20
2.6.2 CUDA Mesaj Geçiş Arayüzü (CUDA-Aware MPI) 20
2.6.3 CUDA Rastgele Sayı Kütüphanesi (CUDA Random Number Generation Library-cuRAND) 21
3. OPTİMİZASYON VE CUDA ÇÖZÜMLERİ **22**
3.1 Optimizasyon 22
3.2 CUDA Kullanılarak Yapılan Metasezgisel Algoritma Çözümleri 24
4. UYGULAMA VE DENEYSEL ÇALIŞMALAR **27**
4.1 Parçacık Sürü Optimizasyon Algoritması 27
4.2 Test Fonksiyonları 28
4.3 Geliştirilen Uygulama 30
4.3.1 PSO CPU Uygulaması 31
4.3.2 PSO Hibrit Uygulaması 34
4.3.3 PSO CUDA Uygulaması 37
4.4 Test Sonuçları 39
5. SONUÇLAR VE DEĞERLENDİRME **53**
6. KAYNAKLAR **54**
ÖZGEÇMİŞ **59**

ŞEKİL LİSTESİ

Sayfa

Şekil 2.1: Paralel hesaplamaya genel örnek. 3
Şekil 2.2: Bordro işleme paralel hesaplama yöntemi. 4
Şekil 2.3: Seri hesaplama genel örneği. 4
Şekil 2.4: Bordro işleme seri hesaplama yöntemi. 5
Şekil 2.5: Flynn sınıflandırmasının basit iç mimarileri. 6
Şekil 2.6: Paralel hızlanma faktörünün elde edilişi. 8
Şekil 2.7: f değerine bağlı olarak değişen hızlanma faktörü. 9
Şekil 2.8: CPU ve GPU genel mimarileri. 14
Şekil 2.9: Block ve Thread yapısı. 17
Şekil 2.10: Kernel, Grid, Block ve Thread yapıları. 17
Şekil 2.11: CUDA bellek yapısı. 19
Şekil 4.1: PSO CPU uygulaması akış diyagramı. 33
Şekil 4.2: Hibrit yaklaşım Thread Block yapısı. 34
Şekil 4.3: Her bir parçacık için tutulan 128 boyutlu belleğin temsili. 35
Şekil 4.4: PSO hibrit uygulaması akış diyagramı. 36
Şekil 4.5: Örnek paralel CUDA Reduction yapısı. 37
Şekil 4.6: PSO CUDA paralel uygulaması akış diyagramı. 38
Şekil 4.7: 64 boyut için CUDA hızlanma grafiği. 47
Şekil 4.8: 64 boyut için hibrit hızlanma grafiği. 48
Şekil 4.9: 128 boyut için CUDA hızlanma grafiği. 49
Şekil 4.10: 128 boyut için hibrit hızlanma grafiği. 50

TABLO LİSTESİ

Sayfa

Tablo 4.1: Tek modlu test fonksiyonları. 29
Tablo 4.2: Çok modlu test fonksiyonları. 30
Tablo 4.3: Kullanılan donanım bilgileri. 31
Tablo 4.4: 64 boyut 1000 iterasyon sonuçları ve hızlanma değerleri. 41
Tablo 4.5: 128 boyut 1000 iterasyon sonuçları ve hızlanma değerleri. 42
Tablo 4.6: 64 boyut 2500 iterasyon sonuçları ve hızlanma değerleri. 43
Tablo 4.7: 128 boyut 2500 iterasyon sonuçları ve hızlanma değerleri. 44
Tablo 4.8: 64 boyut 5000 iterasyon sonuçları ve hızlanma değerleri. 45
Tablo 4.9: 128 boyut 5000 iterasyon sonuçları ve hızlanma değerleri. 46
Tablo 4.10: 64 boyut için CUDA ve hibrit hızlanma değerleri. 51
Tablo 4.11: 128 boyut için CUDA ve hibrit hızlanma değerleri. 51

KISALTMALAR LİSTESİ

API : Application Programming Interface (Uygulama Programlama Arayüzü)
CPU : Central Process Unit (Merkezi İşlemci Birimi)
CUDA : Compute Unified Device Architecture (Birleşik Cihaz Hesaplama Mimarisi)
CUDNN : CUDA Deep Neural Network (CUDA Derin Sinir Ağı)
GPU : Graphics Processing Unit (Grafik İşlemci Birimi)
MPI : Message Passing Interface (Mesaj Geçiş Arayüzü)
MPMD : Multiple Program Multiple Data (Çoklu Program Çoklu Veri)
OPENMP : Open Multi Parallelism (Açık Çoklu Paralellik)
PSO : Particle Swarm Optimization (Parçacık Sürü Optimizasyonu)
SIMD : Single Instruction Multiple Data (Tek Komut Çoklu Veri)
SM : Streaming Multiprocessor (Akış Çok İşlemcili)
SP : Streaming Processor (Akış İşlemcisi)
SPMD : Single Program Multiple Data (Tek Program Çoklu Veri)

ÖNSÖZ

Bu çalışmada, metasezgisel algoritmalardan biri olan parçacık sürü optimizasyon algoritması, seri ve CUDA paralel programlama yaklaşımı ile uygulanmıştır. Tek ve çok modlu test fonksiyonları üzerinde yapılan çalışmanın sonuçları ve hızlanma değerleri incelenmiştir. Öncelikle tez konusu belirlememde yardımcı olan, zorlu pandemi sürecinde her türlü desteklerini çalışmalarımı geliştirme aşamasında esirgemeyen Dr. Öğr. Üyesi Gültekin Kuvat'a sonsuz teşekkürlerimi sunarım. Ayrıca bu süreçte beni maddi ve manevi destekleyen aileme de teşekkürü bir borç bilirim.

Balıkesir, 2022 **Muhammet Taha Aydın**

1. GİRİŞ

Günümüzde kullanım alanı her geçen gün artan optimizasyon yaklaşımları yapay zeka uygulaması geliştirmede özellikle sinir ağlarında doğrusal olmayan problemlerin çözümünde, ekonominin bazı alanlarında [1], meteoroloji alanında hava durum tahmininde [2] ve fabrikalarda iş süreçleri vb. bir çok alanda kullanılmaktadır. Örneğin bir yapay sinir ağında hesaplanacak hatayı en aza indirmek için SGD ve Adam gibi optimizasyon algoritmaları kullanılmaktadır [3].

Optimizasyon algoritmalarının geliştirilmesi ve zamanla kullanım alanlarının arttırılması birçok alanda gözle görünür şekilde fayda sağlayacağı öngörülmektedir. Fakat optimize edilecek problem boyutlarının artması, yeni hesaplama yöntemlerinin ve uygulamalarının yazılımsal ve donanımsal olarak geliştirilmesine yol açmıştır. İlk olarak işlemcilerin hesaplama yapma yeteneklerinin arttırılması ve sürenin kısaltılması amacıyla paralel hesaplama yöntemleri geliştirilmeye başlanmış [4] ve daha sonra bu sürece grafik işlemciler dahil edilmiştir. Bu konudaki en büyük gelişmelerden biri çok Thread'li ve çoklu çekirdek yapısına sahip bütünleşik aygıt işleme mimarisinin (Compute Unified Device Architecture-CUDA) NVIDIA tarafından 2006 yılında tanıtılması olmuştur.

CUDA'nın donanımsal ve yazılımsal olarak geliştirilmesi ve bu mimarinin çok çekirdekli yapısının yeni nesil NVIDIA ekran kartlarının çoğunda olması, paralel hesaplama yöntemlerinin kullanıcılar tarafından geliştirilecek optimizasyon uygulamalarında kolaylıkla kullanılabilirliğini sağlamaktadır. Bu paralel hesaplama yaklaşımı, literatürde sıkça karşılaşılan ve yüksek boyutlu problemlerin çözümünde kullanılan metasezgisel algoritmalarda [5] hesaplama süresinin azaltılması amacıyla kullanılabilir. Bu algoritmalar geliştirilirken ve tasarlanırken genellikle doğada bulunan çok popülasyonlu canlı varlıkların barınma, beslenme ve göç gibi problemlerinden esinlenilmiştir.

Doğadan esinlenilerek geliştirilen metasezgisel optimizasyon algoritmalardan biri de parçacık sürü optimizasyonudur. Basit olarak sürü bilincine ve sezgisine dayanan bir yaklaşımdır. Parçacık sürü optimizasyonu (PSO) bireyler arasındaki bilgi paylaşımına dayanmaktadır. Her birey, parçacık olarak adlandırılmakta olup ve parçacıklardan oluşan nüfusa da sürü (swarm) denilmektedir.

Sürüde bulunan her bir birey, önceki tecrübelerinden yararlanarak ulaşması gereken noktaya göre en iyi konumunu ayarlamaya çalışmaktadır. PSO temelde bir grup veya sürüdeki parçacık pozisyonlarının en iyi noktaya doğru sezgisel ve rastgele gerçekleşen yaklaşımlarla ulaşmasına dayanmaktadır. Bu durumun gerçekleşmesi bahsedildiği üzere rastgele gelişen bir süreçtir ve çoğunlukla bireyler yeni hareketlenmelerinde bir önceki bulunduğu noktadan daha iyi bir konuma gitmeye çalışmaktadır. Bahsedilen bu süreç en iyi noktaya varıncaya kadar veya belirtilen iterasyon sayısı kadar devam etmektedir [6].

Bu tezde bu fikirle yola çıkılarak metasezgisel optimizasyon algoritması olan PSO 64 ve 128 boyutlu problemlerin CPU üzerinde seri, CPU-CUDA hibrit ve CUDA kullanarak paralel çözülmesi üzerine çalışılmıştır. Genellikle çok boyutlu problem çalışmaları için kullanılan 18 farklı test fonksiyonu tercih edilmiştir. Bunlar Rosenbrock, Sphere, Schewefel 2.22, Maximization, Step, Quartic Noise, Griewank, Schwefel 2.20, Schwefel 2.23, Sum Squares, Exponential, Powell, Periodic, Salomon N.1, Xin-She Yang N.2, Alphine N.1, Ackley ve Rastrigin test fonksiyonlarıdır. Hibrit ve CUDA üzerinde uygulama geliştirilirken, parçacık konum ve hız güncelleştirilmesinde paralel yaklaşım uygulanmıştır. Hibrit ve CUDA mimarisi kullanılarak geliştirilen paralel uygulamalarda seri programlama yaklaşımına göre hızlanma tespit edilmiştir.

Bu tez çalışması beş bölümden oluşmaktadır. Birinci bölümde tez ile ilgili genel bilgiler verilmiş, tanıtılmış, önemi ve literatüre sağlayabileceği katkıdan bahsedilmiştir. İkinci bölümde paralel hesaplama yöntemleri açıklanmış, ayrıca çalışmamızda paralel geliştirmede kullanılan CUDA mimarisinin donanım ve yazılım özellikleri anlatılmıştır. Üçüncü bölümde optimizasyon ve CUDA mimarisi kullanılarak yapılan metasezgisel algoritma çalışmalarına yer verilmiştir. Dördüncü bölümde geliştirilen uygulamalar açıklanmış ve test fonksiyonlarıyla yapılan farklı iterasyonlardaki test sonuçları sunulmuştur. Beşinci bölümde ise çalışmanın değerlendirilmesi yapılmış, genel sonuçlardan ve gelecek çalışmalardan bahsedilmiştir.

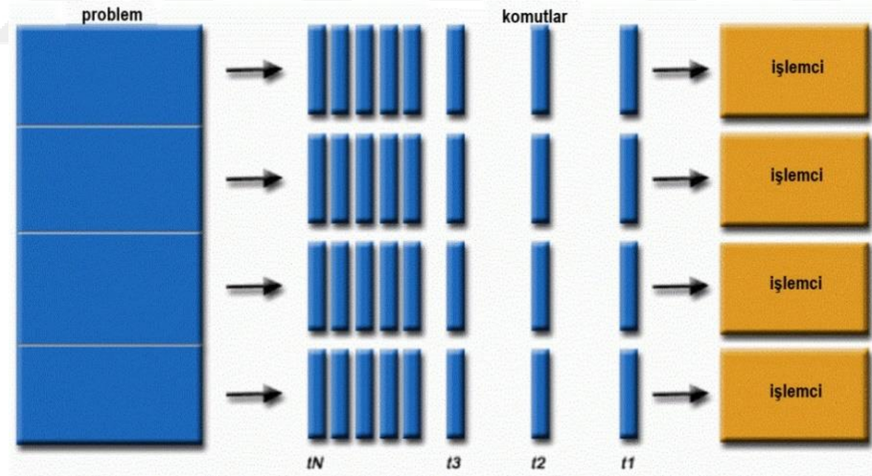
2. PARALEL HESAPLAMA

En temel manasıyla paralel hesaplama, karmaşık hesaplama problemini çözümlemek için birden fazla hesaplama kaynağının eş zamanlı olarak kullanılacak hale getirilmesidir.

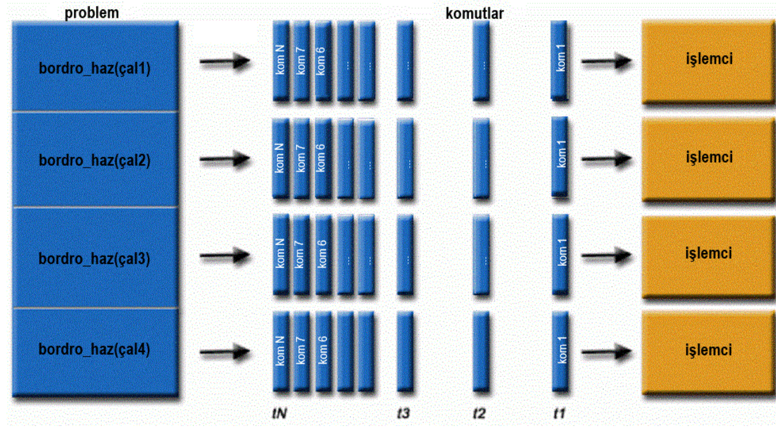
Genellikle kullanım şekli aşağıdaki gibidir [7];

1. Çözümlenecek bir problem, aynı anda çözülebilecek ayrı parçalara bölümlenir.
2. Parçalar daha sonra bir dizi komuta bölünür.
3. Her bir parçadan gelen komutlar, farklı çekirdek veya bilgisayar işlemcilerinde eş zamanlı yürütülür.
4. Genelleyici kontrol/koordinasyon yapısı kullanılmaktadır.

Paralel hesaplama yönteminde bir problem Şekil 2.1'de gösterildiği üzere parçalanır ve eş zamanlı farklı işlemcilerde yürütülür. Örneğin Şekil 2.2'de bordro hesaplaması yapılırken, işlemin daha hızlı yürütülmesi amacıyla her bir çalışanın komutlarının (vergi, saat hesaplama gibi) paralel olarak yürütülmesi gösterilmektedir.



Şekil 2.1: Paralel hesaplamaya genel örnek.

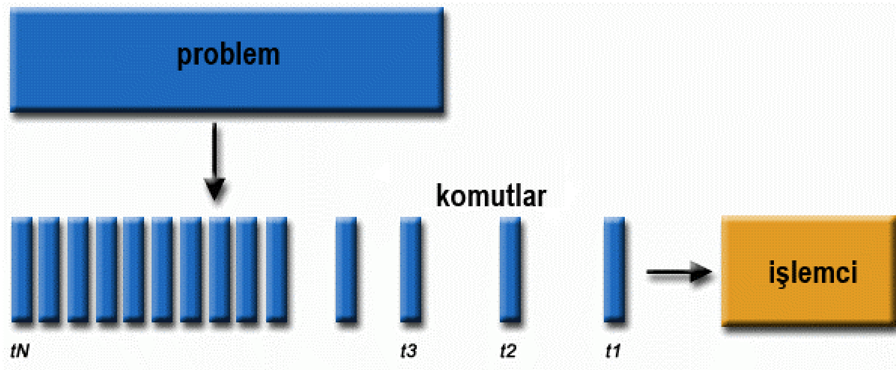


Şekil 2.2: Bordro işleme paralel hesaplama yöntemi.

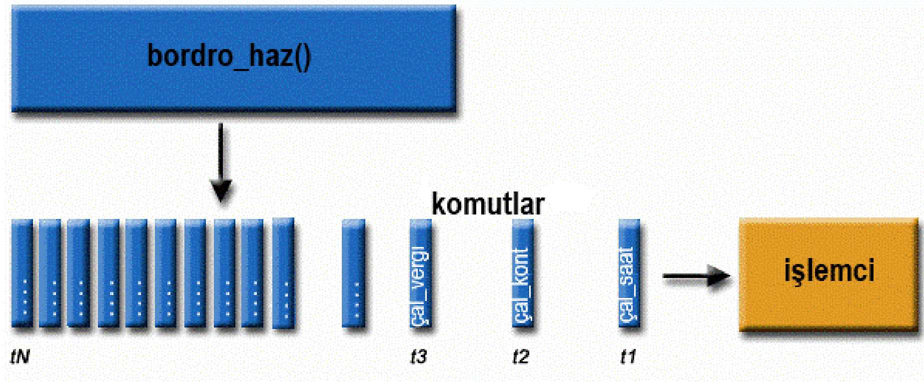
Aynı durum seri hesaplama da aşağıdaki gibidir [7];

1. Bir problem, farklı komut dizisine bölünür.
2. Komutlar sırasıyla işletilir.
3. Bir işlemci üzerinde bu işlem yapılır.
4. Herhangi bir anda yalnızca bir komut yürütülebilir.

Şekil 2.3'te bir problemin seri hesaplama yöntemiyle sıralı zamanlarda işlenmesi görülmekte ve Şekil 2.4'te görüldüğü üzere seri hesaplamalı bordro işleme sürecindeki her işlem paralel hesaplamadaki gibi eş (çalışan vergi, saat hesabı vb.) zamanlı değil sıralı farklı zamanlarda yürütülmektedir [7].



Şekil 2.3: Seri hesaplama genel örneği.



Şekil 2.4: Bordro işleme seri hesaplama yöntemi.

Yaşadığımız gerçek dünyada, birçok kompleks, birbiriyle alakalı olaylar aynı anda, ancak zamansal sıra içinde gerçekleşmektedir. Sıralı hesaplama ile karşılaştırıldığında, paralel hesaplama kompleks, ancak gerçek dünya problemlerini çözümlemek, modelini oluşturmak, problemlerin benzer durumlarını simüle etmek ve anlamak için çok daha uygundur. Bu yüzden genellikle karmaşık hesaplama yöntemlerinin gerektiği, astronomide, iklim değişim hesaplamalarında, tektonik hareket hesapları gibi durumlarda paralel programlama kullanılmaktadır [7].

2.1 Von Neumann Bilgisayar Mimarisi

Adını 1945 tarihli makalelerinde elektronik bir bilgisayar için genel gereksinimleri ilk kez yazan Macar matematikçi John Von Neumann'dan almıştır. "Depolanmış program bilgisayarı" olarak da bilinir. Hem program talimatları hem de veriler elektronik bellekte tutulmaktadır. "Sabit kablolama" ile programlanmış önceki bilgisayarlardan farklıdır.

Dört ana bileşenden oluşmaktadır [7];

- Hafıza
- Kontrol Ünitesi
- Aritmetik Mantık Birimi
- Giriş/Çıkış

Okuma/yazma, rastgele erişimli bellek hem program talimatlarını hem de verileri depolamak için kullanılmaktadır. Program talimatları, bilgisayara bir şey yapmasını söyleyen kodlanmış verilerdir. Veriler sadece program tarafından kullanılacak bilgilerdir. Kontrol ünitesi hafızadan talimatları/verileri alır, talimatların kodunu çözmekte ve ardından programlanmış

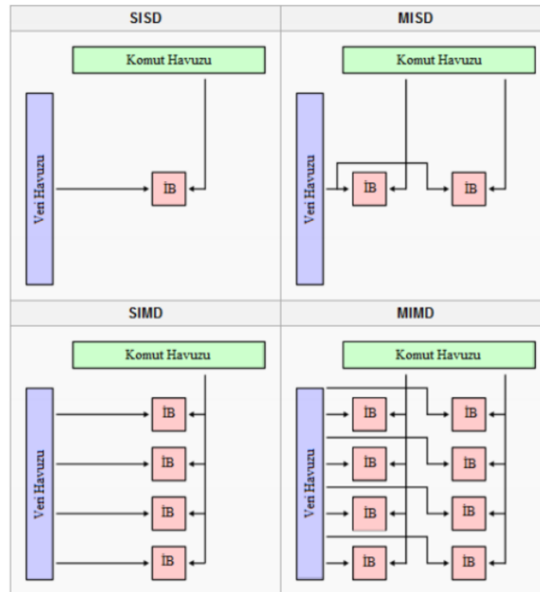
görevi gerçekleştirmek için işlemleri sırayla koordine etmektedir. Aritmetik Birimi temel aritmetik işlemleri gerçekleştirir. Giriş/Çıkış, insan operatörün ara yüzüdür. Günümüzde paralel bilgisayarlar genellikle temelde bu mimariyi kullanmaya devam etmektedir.

Paralel bilgisayarları sınıflandırmanın birkaç farklı yolu vardır. 1966'dan beri kullanımda olan ve daha yaygın olarak kullanılan sınıflandırmalardan biri Flynn sınıflandırması (taksonomisi) olarak adlandırılır. Flynn sınıflandırması, çok işlemcili bilgisayar mimarilerini talimat akışı ve veri akışı olmak üzere iki bağımsız boyuta göre nasıl sınıflandırılabileceklerine göre ayırmaktadır. Bu boyutların her biri, iki olası durumdan yalnızca birine sahip olabilir; tekli veya çoklu [7].

Flynn'e göre 4 olası sınıflandırma görülmektedir;

- Tek komut, tek veri
- Tek komut, çoklu veri
- Çoklu komut, tek veri
- Çoklu komut, çoklu veri

Sınıflandırmanın basit iç mimarileri Şekil 2.5'te verilmiştir [7].



Şekil 2.5: Flynn sınıflandırmasının basit iç mimarileri.

Tek Komut Tek Veri:

Tek Komut; Herhangi bir saat döngüsü sırasında CPU tarafından yalnızca bir komut akışı yürütülmektedir.

Tek Veri; Herhangi bir saat döngüsü sırasında giriş olarak yalnızca bir veri akışı kullanılmaktadır.

Tek Komut Çoklu Veri:

Tek Komut; Bütün işlem birimleri, aynı saat döngüsünde aynı talimatı yürütür.

Çoklu Veri; Her bir işlem birimi farklı bir veri öğesi üstünde çalışabilmektedir.

Grafik/görüntü işleme gibi yüksek derecede düzenlilikle karakterize edilen özel problemler için en uygunudur. Çoğu modern bilgisayar, özellikle grafik işlemci birimlerine sahip olanlar, SIMD talimatlarını ve yürütme birimlerini kullanılmaktadır.

Çok Komut Tek Veri:

Çoklu Talimat; her işlem birimi, ayrı talimat akışları aracılığıyla veriler üzerinde bağımsız olarak çalışmaktadır.

Tek veri; Tek bir veri akışı, birden çok işlem birimine beslenmektedir.

Çoklu Komut Çoklu Veri:

Çoklu Talimat; Her işlemci farklı bir talimat akışı yürütüyor olabilmektedir.

Çoklu Veri; Her işlemci farklı bir veri akışıyla çalışıyor olabilmektedir.

Yürütme senkronize veya asenkron, deterministtik veya deterministtik olmayan olabilir. Şu anda, en yaygın paralel bilgisayar türü çoğu modern süper bilgisayar bu kategoriye girmektedir [7].

2.2 Hızlanma Faktörü

Problemlerin zorlaşması ve buna bağlı olarak çözüm sürelerinin uzaması, araştırmacıları paralel yöntemlerin kullanımına yöneltmektedir. Paralel yöntemlerin başarısı hızlanma faktörü ile ölçülmektedir. Hızlanma faktörü, bir işin seri çözüm süresinin paralel çözüm süresine bölünmesi ile hesaplanır. Hızlanma faktörü (S) basit hali ile Denklem 2.1'deki gibi hesaplanmaktadır [8].

$$S = \frac{t_s}{t_p} = \frac{Seri\ yöntemde\ harcanan\ süre}{Paralel\ yöntemde\ harcanan\ süre} \tag{2.1}$$

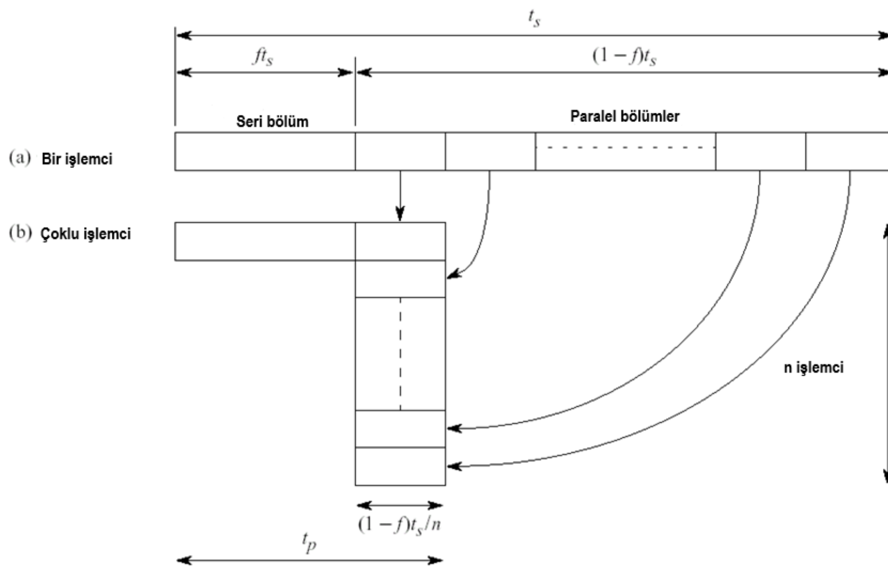
Denklem 2.1'de S hızlanma faktörü, t_s seri yöntemde harcanan süre ve t_p paralel yöntemde harcanan süreyi ifade etmektedir. Dolayısıyla t_p değeri ne kadar düşük olursa hızlanma o kadar yüksek olacaktır. Hızlanmanın yüksek olması, paralel yöntemin başarı ve etkisini göstermektedir.

Paralel bir mimarinin hızlanmaya etkisi donanım gücüne bağlı olduğu gibi problemin yapısına da bağlıdır. Bir problemin bazı kısımları paralel olarak bölünemeyebilir. Buna bağlı olarak istenildiği kadar yüksek bir hızlanma elde edilemeyebilir. Bu durum Amdahl Kanunu ile ortaya konmuştur.

Şekil 2.6'da [8] gösterildiği gibi bir problemin seri kısmı ne kadar küçük ise paralel kısmı o kadar büyüktür. Burada kullanılan f değeri, çözülen problemde paralel olarak bölünemeyen kısmın oranını ifade etmektedir. Amdahl Kanunu'na göre t_s ve t_p değerleri Denklem 2.2 ve Denklem 2.3'teki gibi hesaplanabilir. Denklem 2.3'te bulunan n ifadesi paralel yönteme göre problemin kaç parçaya bölündüğünü veya işlemci sayısını göstermektedir [8].

$$t_s = ft_s + (1-f)t_s \tag{2.2}$$

$$t_p = ft_s + \frac{(1-f)t_s}{n} \tag{2.3}$$

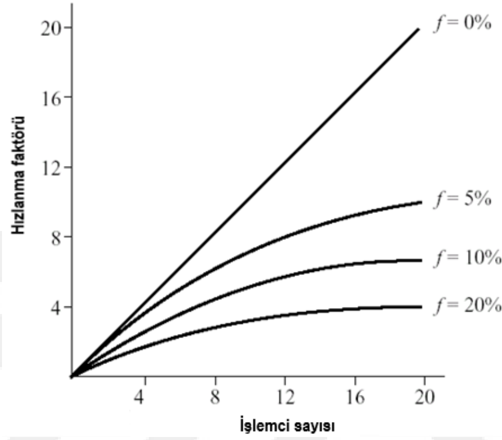


Şekil 2.6: Paralel hızlanma faktörünün elde edilişi.

Amdahl Kanunu'na göre hızlanma faktörü Denklem 2.4'teki gibi [8] hesaplanabilir.

$$S = \frac{t_s}{t_p} = \frac{t_s}{f t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1 + (n-1)f} \quad (2.4)$$

Şekil 2.7'de görüldüğü gibi f değeri arttıkça problemin seri kısmından gelecek zorunlu zaman yükü artacağından hızlanma azalmaktadır [8].



Şekil 2.7: f değerine bağlı olarak değişen hızlanma faktörü.

2.3 Paralel Bilgisayar Bellek Mimarisi

Bu bölümde paralel hesaplamada kullanılan bellek mimarilerinden bahsedilmektedir.

Paylaşımlı Bellek: Paylaşımlı bellek paralel bilgisayarlar çok çeşitlidir, fakat temel olarak işlemcilerin temel adres alanı olarak bütün belleğe erişme yeteneğine sahiptirler. Birden fazla işlemci diğerlerine bağımlı kalmaksızın çalışabilmekte fakat aynı bellek kaynaklarını paylaşabilmektedir. Bir işlemci tarafından yürütülen bir bellek pozisyonundaki farklılıklar diğer işlemcilerden de görülebilmektedir.

Dağıtık Bellek: Dağıtık bellek yapıları, işlemciler arası belleği bağlamak için bir sistem ağına ihtiyaç duymaktadır. İşlemcilerin kendi lokal bellekleri mevcuttur. Bir işlemcideki bellek adresleri başka bir işlemci tarafından erişilemez. Bu nedenle tüm işlemcilerde genel adres alanı durumu söz konusu değildir. Her işlemcinin kendi lokal belleği olduğundan diğerlerine bağımlı olmaksızın çalışmaktadır. Lokal belleğinde yaptığı değişikliklerin diğer

işlemcilerin belleği üzerinde hiçbir etkisi yoktur. Buna bağlı olarak, önbellek tutarlılığı durumu mevcut değildir. Bir işlemcinin başka bir işlemcideki verilere ulaşması gerektiğinde, verilerin nasıl ve ne zaman iletileceğini açıkça tanımlamak genellikle programı yazan kişinin görevidir.

Hibrit Paylaşımlı-Dağıtık Bellek: Günümüzde en büyük ve en güçlü bilgisayarları hem paylaşımlı hem de dağıtık bellek yapılarını kullanmaktadır. Paylaşımlı bellek birleşeni, paylaşılan bir bellek bilgisayarı veya grafik işlemci birimleri (GPU) olabilir. Dağıtık bellek birleşeni, başka bir bilgisayardaki belleği değil, yalnızca kendi belleği hakkında bilgi sahibi olan birden çok paylaşılan GPU bellek ağıdır. Bu sebeple, verileri bir bilgisayardan diğerine iletmek için ağ yapısı gerekmektedir. Mevcut yönelimler, bu tür bellek yapılarının gelecek için en üst düzeyde bilgi işleminde hüküm sürmeye ve artmaya devam edeceğini göstermektedir [7].

2.4 Paralel Programlama

Paralel programlama, bilimsel ve teknolojik ilerlemelerle ortaya çıkan kompleks ve çözülmesi zor problemlerin çözüme kavuşturulmasında, problemlerin farklı taraflarını farklı işlemcilere bölerek gereksinim duyulan veri işleme hızı ve verimliliğin sağlanmasında yararlı olmaktadır.

Paralel programlamada bir problemin çözümü aşağıdaki sıralı yöntemler ile gerçekleştirilmektedir;
1) Problem daha alt yapılara parçalanır.
2) Bu parçalar eşit zaman paydalarına bölünmektedir.
3) Her problem bölümü farklı işlemcilerde, aynı anda eş zamanlı olarak işleme verilir ve programlama yaklaşımı gerçekleştirilmiş olmaktadır.

Paralel program yaklaşımının en basit tercih sebepleri bellek kullanımını en verimli şekilde yapılandırarak bilgisayardaki gecikmelerin önüne geçilmesidir. Bu sayede bilgisayarlar hesaplamaları daha süratli bir şekilde yapabilmekte ve hızlanma sağlanmaktadır [9] .

Genel kullanımda olan programlama modelleri;

- Paylaşımlı Bellek (İş Parçacığı Olmadan)

- İş Parçacıkları
- Dağıtık Bellek / Mesaj Geçiş
- Paralel Veri Modeli
- Hibrit Model
- Tek Program Çoklu Veri
- Çoklu Program Çoklu Veri

Paylaşımlı Bellek: Bu programlama modelinde süreçler/görevler, eş zamansız olarak okuyup yazdıkları ortak bir adres alanını paylaşmaktadır. Paylaşımlı belleğe erişimi kontrol etmek, çekişmeleri çözmek, yarış koşullarını ve kilitlenmeleri önlemek için kilitler/semaforlar gibi çeşitli mekanizmalar kullanılmaktadır. Bu belki de en basit paralel programlama modelidir. Programcının bakış açısından bu modelin bir avantajı, veri sahipliği kavramının eksik olmasıdır, bu nedenle görevler arasında veri iletişimini açıkça belirtmeye gerek yoktur. Tüm işlemler paylaşımlı belleği görür ve eşit erişime sahiptir [7].

İş Parçacıkları: Bu programlama modeli, bir tür paylaşımlı bellek programlamasıdır. Paralel programlamanın Thread modelinde, tek bir ağırlıklı işlem, birden çok hafif, eşzamanlı yürütme yoluna sahip olabilmektedir [7]. Bazı iş parçacık uygulamalarına; Python, Java, Microsoft, GPU'lar için CUDA iş parçacıkları örnek olarak verilebilir.

Dağıtık Bellek: Hesaplama sırasında kendi yerel belleğini kullanan bir dizi görev atanır. Aynı fiziksel makinede ve/veya rastgele sayıda makinede birden fazla görev bulunabilmektedir. Görevler, mesaj gönderip alarak iletişim yoluyla veri alışverişinde bulunabilmektedir. Veri aktarımı genellikle her bir işlem tarafından gerçekleştirilecek ortak işlemler gerektirmektedir.

Tarihsel olarak, 1980'lerden beri çeşitli mesaj ileten kütüphaneler mevcuttur. Bu uygulamalar birbirinden önemli ölçüde farklıydı ve bu da programcıların taşınabilir uygulamalar geliştirmesini zorlaştırıyordu. 1992'de MPI (Mesaj Geçiş Arayüzü – Message Passing Interface) forumu, mesaj iletme uygulamaları için standart bir arayüz oluşturma birincil amacı ile kurulmuştur. MPI, neredeyse tüm diğer mesaj iletme uygulamalarının yerini alarak mesaj iletimi için fiili endüstri standardı haline gelmiştir. Neredeyse tüm

popüler paralel hesaplama platformları için MPI uygulamaları mevcut olarak bulunmaktadır [7].

Paralel veri modeli aşağıdaki özellikleri gösterir:

- Adres alanı global olarak işlenmektedir.
- Paralel çalışmanın çoğu, bir veri kümesi üzerinde işlem gerçekleştirmeye odaklanmaktadır. Veri seti tipik olarak bir dizi veya küp gibi ortak bir yapı halinde düzenlenmektedir.
- Bir dizi komut aynı veri yapısı üzerinde toplu olarak işlenir, ancak her komut aynı veri yapısının farklı bir bölümünde çalışmaktadır.
- Paylaşımlı bellek mimarilerinde, tüm görevler global bellek aracılığıyla veri yapısına erişilebilmektedir.
- Dağıtık bellek mimarilerinde, global veri yapısı görevler arasında mantıksal ve/veya fiziksel olarak bölünebilmektedir.

Hibrit Model: Farklı paralel hesaplama yaklaşımlarının bir arada kullanıldığı modellerdir. Hibrit modelin yaygın bir örneği olarak, mesaj geçiş modelinin iş parçacığı modeli ile birleşimi olarak verilebilmektedir. İş parçacıkları tarafından, yerel, düğüm üzerindeki verileri kullanarak hesaplama açısından yoğun işlemler gerçekleştirilirken farklı düğümlerdeki süreçler arasındaki iletişim, MPI kullanılarak ağ üzerinden gerçekleşmektedir. Bu hibrit model, kümelenmiş çok çekirdekli makinelerin donanım ortamına çok uygundur.

Bir başka benzer ve giderek daha popüler hale gelen hibrit model örneği, MPI'ı CPU-GPU programlama ile kullanmaktır. MPI görevleri, yerel belleği kullanan ve bir ağ üzerinden birbirleriyle iletişim kuran CPU'larda çalışır. Hesaplama açısından yoğun işler, düğümlerdeki (nodes) GPU'lara yüklenmektedir. Yerel düğüm belleği (local node memory) ve GPU'lar arasındaki veri alışverişinde için CUDA kullanılmaktadır.

Yaygın olan diğer hibrit modeller aşağıdadır [7]:

- PThreads ile MPI
- GPU olmayan hızlandırıcılara sahip MPI

Tek Program Çoklu Veri: Tek program çoklu veri (Single Program Multiple Data- SPMD) aslında diğer bölümde anlatılan paralel programlama yaklaşımlarının herhangi bir varyasyonu üzerine inşa edilen yüksek seviyeli bir programlama yaklaşımıdır.
Tek program; Tüm komutlar aynı programın kopyalarını eş zamanlı yürütür. Bu program iş parçacıkları, mesaj geçişi, veri paralel veya hibrit olabilmektedir.
Çoklu veri; Tüm görevler farklı veriler kullanılabilmektedir.

SPMD programları çoğunlukla, farklı komutların dallanmasına veya programın sadece yürütmek üzere tasarlandığı bölümlerini bir şarta bağlı olarak yürütmesine imkan vermek için programlanmış mantık yapısına sahiptir. İleti geçişini veya hibrit programlamayı kullanan SPMD yaklaşımı, muhtemelen çok düğümlü kümeler için en yaygın olarak kullanılan paralel programlama modelidir.

Çoklu Program Çoklu Veri: SPMD gibi, çoklu program çoklu veride (Multiple Program Multiple Data- MPMD) aslında daha önceki bölümlerde bahsi geçen paralel programlama yaklaşımlarının herhangi bir varyasyonu üzerine inşa edilebilen yüksek seviyeli bir programlama yaklaşımıdır.
Çoklu program; Komutlar eş zamanlı farklı programları çalıştırabilir. Programlar Thread, mesaj geçişi, veri paralel veya hibrit olabilmektedir.
Çoklu veri; Tüm görevler farklı veriler kullanılabilmektedir [7].

2.5 CUDA Mimarisi ve Programlaması

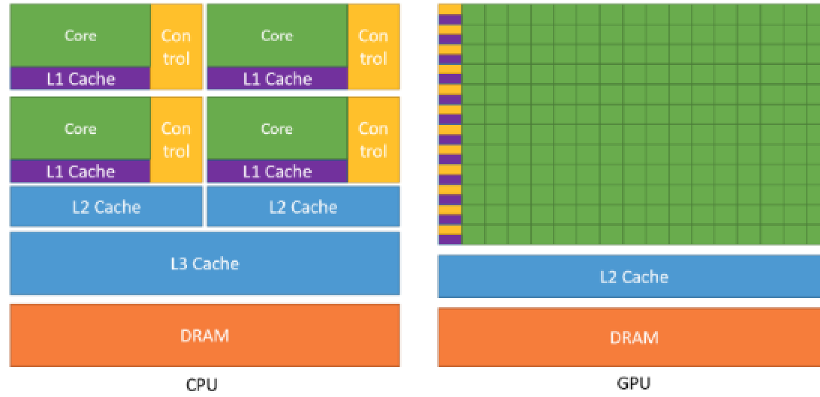
Bu bölümde, uygulama aşamasında kullanılan CUDA'nın donanım ve yazılım mimarisi anlatılmaktadır.

2.5.1 CUDA Donanım Mimarisi

GPU yüksek komut verimi ve bellek bant genişliğinden dolayı CPU'lardan daha hızlı işlem gücüne sahiptir. GPU ve CPU arasındaki bu yetenek farkı, farklı hedefler düşünülerek tasarlandıkları için vardır. CPU, Thread adı verilen bir dizi işlemi paralel olarak yürütebilirken, GPU binlerce parçayı paralel olarak yürütmek üzere tasarlanmıştır.

GPU, paralel hesaplamalar için uzmanlaşmıştır ve bu nedenle, veri önbelleğe alma ve akış kontrolü yerine veri işlemeye daha fazla transistörün ayrılacağı şekilde tasarlanmıştır.

Şekil 2.8'de GPU ve CPU genel mimarileri verilmektedir. Şekilde görüldüğü üzere NVIDIA GPU üzerinde çok fazla işlem çekirdeği bulunmaktadır [10].



Şekil 2.8: CPU ve GPU genel mimarileri.

Genel olarak, bir uygulama paralel parçaların ve sıralı parçaların bir karışımına sahiptir. Bu nedenle sistemler genel performansı en üst düzeye çıkarmak için bir GPU ve CPU karışımı ile tasarlanmıştır. Yüksek derecede paralelliğe sahip uygulamalar, CPU'dan daha yüksek performans elde etmek için GPU'nun paralel doğasından yararlanmaktadır [11].

Kasım 2006'da NVIDIA, zor problemleri CPU'dan daha hızlı çözmek için NVIDIA GPU'lardaki paralel hesaplama motorundan yararlanan, genel amaçlı bir paralel bilgi işlem platformu ve programlama modeli olan CUDA'yı tanıttı. CUDA, geliştiricilerin C++'ı üst düzey bir programlama dili olarak kullanmalarına izin veren bir yazılım ortamıyla birlikte gelir. FORTRAN, Direct Compute, OpenACC gibi diğer diller, uygulama programlama arabirimleri veya direktiflere dayalı yaklaşımları desteklenmektedir. CUDA, NVIDIA grafik işlemci birimlerinde paralel uygulama geliştirme veya grafik işleme işlemlerinin hızlandırılması için oluşturulmuş mimari ve yazılım uygulama programlama arayüzüdür (Application Programming Interface-API). CUDA, programlama dili olarak düşük seviye C programlama dilini baz almıştır. Merkezi işlemcide C dili ile kodlanmış mevcut programları veya geliştirilmiş uygulamaları, GPU üzerinde çoklu Thread kullanılarak çalıştırılmasına imkan sağlamaktadır.

CUDA donanımsal olarak incelendiğinde Thread'ler akış işlemcisi (Streaming Processor-SP) üstünde yürütülmektedir. Çoklu SP işlemcilerinin ortaya çıkardığı yapıya ise akış çoklu

işlemcisi (Streaming Multiprocessor-SM) adı verilmektedir. SM'ler grafik işlemci donanımındaki işlem noktalarını oluşturmaktadır.

2.5.2 CUDA Yazılım Mimarisi

CUDA yazılım mimarisinde donanım seviyesinde çalışan paralel yapılar mevcuttur. Bu yapıların en başında Grid, Block ve Thread yapıları gelmektedir.

İş parçacıkları belli bir düzende problem boyutuna ve zorluğuna göre değişecek şekilde belirlenen Block'lara yerleştirilir ve Block'larda uygun Grid'lerde yer almaktadır. Fakat CUDA mimarisinde çalışacak bir kod bloğu direkt GPU üzerinde yürütülemez. İlk olarak komut veya veriler CPU üzerinde oluşturulur ve paylaşımlı bellek aracılıyla bu komut veya veriler GPU üzerine aktarılarak çalıştırılması sağlanmaktadır. Çalışma süreci bittiğinde oluşan veri veya hesaplamalar aynı paylaşımlı bellek üzerinden CPU'ya aktarılır ve kullanıcıya veya programcıya çıktılar sunulur [11] .

2.5.2.1 CUDA Kernel

Geliştirilmiş kodun seri haldeki halinin klonlanarak CUDA üzerinde paralel şekilde çalışmasını ilk tetikleyen yapılara Kernel denilmektedir. GPU, veri grubunun her biri için birer Kernel klonu meydana getirir. Bu Kernel klonları "Thread" olarak adlandırılmaktadır. Kernel komutunun ilk çağrısı CPU tarafında oluşturulur, daha sonra buradan GPU tetiklenerek işlem buraya taşınır. Bu yapı global kod yapısı kullanılarak gerçekleştirilmektedir. Global yapı CPU ve GPU'nun programlama düzeyince haberleşmesini sağlayan yapıdır. Global yapı üzerinde oluşturulan Thread'ler bir araya gelerek Thread dizilerini oluşturmaktadır [11].

2.5.2.2 CUDA Thread

Thread, CUDA'nın ana yapılarından birini oluşturan en küçük programlama parçacığıdır. Bu Thread'ler daha sonra mimaride mevcut olan Block'ların içerisine en fazla üç boyutlu olacak şekilde yerleştirilirler. Kernel oluşturulurken parametre olarak verilen Thread sayıları kadar ve buna bağlı Block sayısına göre eş zamanlı kod yürütülmektedir. İş parçacıklarını belirlenen Block'lar içerisinde indekslenecek şekilde gruplanmaktadırlar. Birbirinden ayrı Block gruplarında bulunan Thread'ler ortak şekilde yürütülmezler. Her bir Thread'in Block içerisinde belirlenen kendisine ait bir kimliği ve belirtildiği üzere konumunu ve ayrıldığı yeri

belirten indeksi mevcuttur. Genellikle Block'larda bulundukları indeksler "ThreadIdx.x" şeklinde ifade edilmektedir[11] .

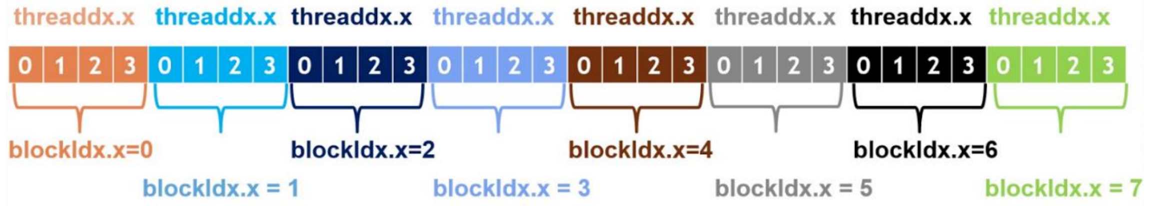
2.5.2.3 CUDA Block

Block'lar, paralel yapıda yürütülen Thread'lerin bir araya getirdiği ve yerleştirildiği yapılardır. Her bir Grid içerisinde tek halde bulunurlar ve Grid içerisinde en fazla üç boyutlu olacak şekilde yerleştirilebilirler. Aynı iş parçacıklarının Block içerisinde olduğu gibi her bir Block için Grid içerisinde kendisi için mevcut indeksleri vardır. Bu indeksler en basit bir boyutlu düzeyde "BlockIdx.x" şeklindedir. Block'lar içerisinde bulundurdukları Thread'in sayısına ve boyutlarına göre boyut kazanmaktadırlar. Bir boyutlu Block düşünüldüğünde bu Block "BlockDim.x" şeklinde temsil edilmektedir. Block boyutu terimi aslında Block'ta bulunan Thread sayısı terimiyle aynı anlama gelir ve Block'ta mevcut Thread sayısı anlamını karşılamaktadır [11].

Aşağıdaki kod parçacığında fonksiyon adı verilecek bir Kernel oluşturulmuş. Kernel oluşturma "<<< >>>" şeklinde yapılmaktadır. Kernel oluşturulurken çağırılacak fonksiyonun parametreleri "(d_a,d_b)" şeklinde parantez içerisinde verilmektedir [11].

```
_global__ void add_array(int *array_a, int * array_b, int *array_c)
{
int index = ThreadIdx.x + BlockIdx.x * BlockDim.x;
array_c[index] = array_a[index] + array_b[index];
}
Threads_per_Block = 4;
no_of_Blocks = 8;
add_array<<<no_of_Blocks,Threads_per_Block>>>(d_a,d_b,d_c);
```

Bu kod parçacığının Block ve Thread yapısı aşağıdaki Şekil 2.9'daki gibidir [11].

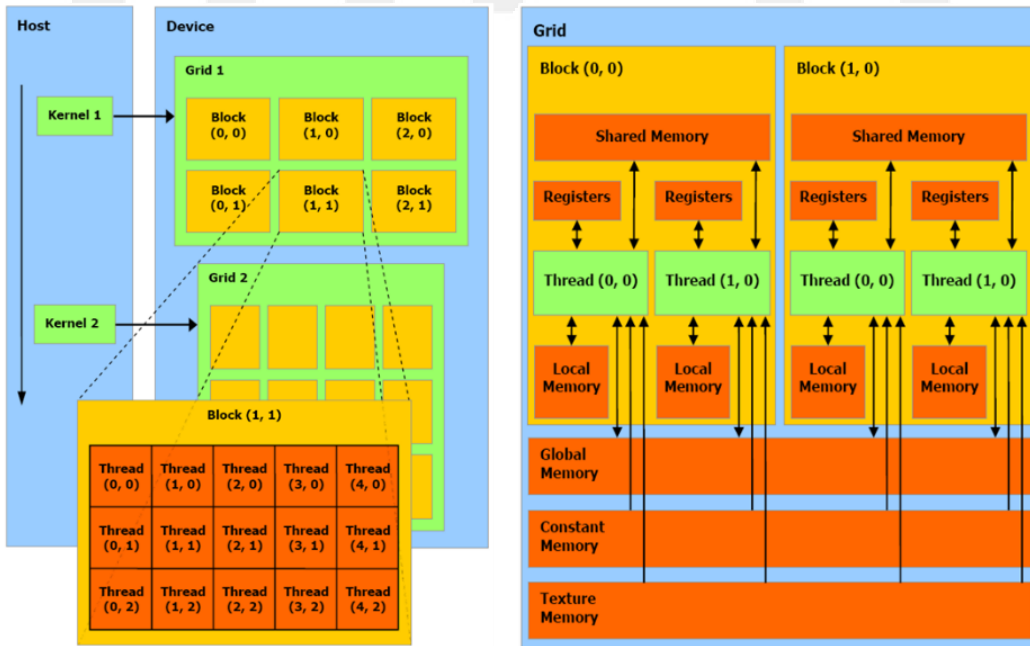


Şekil 2.9: Block ve Thread yapısı.

2.5.2.4 CUDA Grid

Grid, Block'ların belirli indeks düzeninde oluşturdukları yazılımsal yapılardır. Oluşturulan Kernel sayısına bağlı olarak Grid'ler oluşmaktadır. Açıklamak gerekirse paralel oluşturulan her bir Kernel klonu aslında Grid meydana getirmiş olmaktadır. Grid'ler en fazla iki boyutlu olmaktadır. Hesaplama özelliği 2.0 veya daha gelişmiş olan CUDA mimarilerinde veya grafik işlemcilerinde Grid'ler üç boyutlu olabilmektedir. Bu boyutlar bir boyutlu Grid için "GridDim.x" şeklinde temsil edilebilmektedir. Block yapılarında olduğu gibi Grid boyutu aslında bir Grid'de bulunan Block sayısı terimine denktir.

Şekil 2.10'da Kernel, Grid, Block, Thread yapıları ve bellek geçişleri gösterilmektedir [12].



Şekil 2.10: Kernel, Grid, Block ve Thread yapıları.

2.5.2.5 CUDA Lokal Bellek (Local Memory)

CUDA üzerinde program çalışırken programcı için kullanıma izin verilmiş bir bellek tipi değildir. Bu bellek yapısında mevcut veriler iş parçacıklarının işleri bitinceye kadar süre zarfında muhafaza edilir. Her Thread'in kendine özgü bir bellek yapısı mevcuttur ve sadece o Thread'e aittir. Başka indekslerde bulunan Thread'ler bu belleklere ulaşamazlar ve paylaşım söz konusu değildir.

2.5.2.6 CUDA Paylaşımlı Bellek (Shared Memory)

CUDA paylaşımlı bellek, tüm çekirdek gruplarında mevcuttur. Yerel belleğin aksine aynı grupta bulunan iş parçacıkları bu bellek üzerinden birbirleri ile haberleşebilmekte ve yazma okuma işlemleri yapabilmektedirler. Aynı Block yapısında bulunan Thread'ler aynı anda bu bellek noktasına erişmeye çalışmazlarsa hızlı bir şekil okuma yazma işlemi yapılabilmektedir.

2.5.2.7 CUDA Global Bellek (Global Memory)

Block'lar arasında yürütülen her Thread'in erişebildiği, veri yazabildiği ve okuyabildiği genel erişimli bellektir. Host tarafından gelecek herhangi bir komut veya veri GPU üzerine aktırılırken bu bellek kullanılmaktadır. Bu yüzden bu bellek uzun süre aktif olup veri iletişimi aşamasında erişme zamanı bakımından uzun süreler ayakta kalmaktadır.

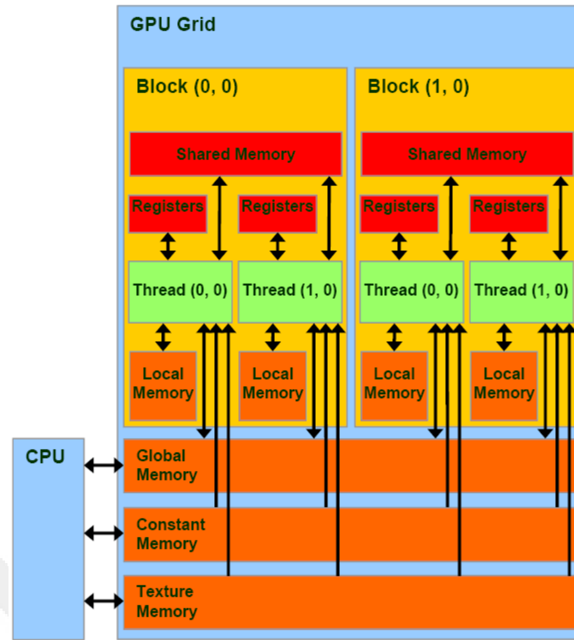
2.5.2.8 CUDA Sabit Bellek (Constant Memory)

Sadece okuma yapılabilen 64 KB boyuta sahip bir bellektir. Kernel oluşturulduğunda üzerinde herhangi bir değişiklik yapılmayan veriler için avantaj sağlamaktadır. Genellikler uygulama geliştirme aşamasında veya program çalıştırılırken herhangi bir yazma işleminin veya veri güncelleme işleminin olmadığı durumlarda kullanılmaktadırlar.

2.5.2.9 CUDA Doku Bellek (Texture Memory)

Bu bellek yapısı sabit bellek yapısıyla benzer olup sadece okuma işlemlerinin yapılabildiği bir bellek yapısıdır. Bu sayede verimlilik ve hız artmakta, veri geçiş trafiği büyük ölçüde azalabilmektedir. Bu yapı genellikle OpenGL veya DirectX için tasarlanmış olup oyun veya grafik tasarımının gerektiği yapılarda kullanılmaktadır. Sabit bellek gibi hızlı şekilde çalışabilmek amacıyla çerez yapısına sahiptir. Bazı yenileme (render) işlemlerinde bu yapı daha az veri yoğunluğunda verimli düzeyde bant genişliği sağlamaktadır.

Şekil 2.11'de GPU içerisindeki Thread bellekleri ve GPU bellekleri gösterilmektedir [12].



Şekil 2.11: CUDA bellek yapısı.

Bazı cuda bellek yönetim fonksiyonları;

```
cudaMalloc (void ** pointer, size_t nbytes);
```

C dilindeki Malloc ile aynıdır. Belirtilen byte kadar GPU belleğinden yer ayırır.

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

İstenilen değer, istenildiği kadar bellek alanına yerleştirilebilir.

```
cudaFree (void* pointer);
```

C dilindeki Free ile eşdeğerdir, ayrılan bellek serbest bırakılır.

```
cudaMemcpy (void *dst, const void *src, size_t count,
enum, cudaMemcpy kind);
```

“cudaMemcpy” bellekteki verinin kopyalama işlemleri için kullanılır. “dst” hedef bellek adresini, “src” kaynak bellek adresini, “count” kopyalanacak byte sayısını, “kind” ne tür bir iletim olacağını temsil etmektedir.

“kind” yani iletim türü 4 farklı şekilde olabilmektedir;

“cudaMemcpyHostToHost”	Host → Host (CPU → CPU)
“cudaMemcpyHostToDevice”	Host → Device (CPU → GPU)
“cudaMemcpyDeviceToHost”	Device →Host (GPU → CPU)
“cudaMemcpyDeviceToDevice”	Device → Device (GPU → GPU)

2.6 CUDA Uygulama Alanları

Bu bölümde CUDA’nın kullanıldığı veya geliştirildiği uygulama alanları anlatılmaktadır. Metasezgisel algoritmaların paralel olarak modellenmesiyle geliştirilen CUDA uygulamalarına başka başlık altında bahsedilmiştir.

2.6.1 CUDA Derin Sinir Ağı (CUDA Deep Neural Network-cuDNN)

CUDA Derin Sinir Ağı kütüphanesi (cuDNN) derin sinir ağları için GPU ile hızlandırılmış bir kütüphanedir. cuDNN, ileri ve geri konvolüsyon (back-forward convolution), havuzlama (pooling), normalizasyon ve aktivasyon katmanları gibi standart rutinler için uygulamalar sunar.

Dünya çapındaki derin öğrenme araştırmacıları ve çerçeve geliştiricileri, yüksek performanslı GPU hızlandırması için cuDNN kullanmaktadır. cuDNN düşük seviye GPU performans ayarlaması için zaman harcamak yerine sinir ağları eğitmeye ve yazılım uygulamaları geliştirmeye odaklanmayı sağlamaktadır. cuDNN, Caffe, Caffe2, Chainer, Keras, MATLAB, MxNet, TensorFlow ve PyTorch dahil olmak üzere yaygın olarak kullanılan derin öğrenme çerçevelerini hızlandırmaktadır [13].

2.6.2 CUDA Mesaj Geçiş Arayüzü (CUDA-Aware MPI)

Mesaj Geçiş Arayüzü (Message Passing Interface-MPI), çok bilgisayar ve iş istasyonunun olduğu bir ağ yapısında bir programın birçok makinede çalıştırılarak birbirleri arasında veri iletişimi kurulmasını sağlayan bir kütüphanedir. Son gelişmelerle birlikte MPI, tek bir bilgisayarda veya düğümde paralel hesaplama için tasarlanmış olan CUDA ile tam olarak uyumlu hale gelmiştir. MPI ve CUDA’nın iki paralel programlama yaklaşımını birleştirmek

istenmesinin birçok nedeni vardır. Yaygın bir neden, tek bir GPU'nun belleğine sığmayacak kadar büyük ya da tek bir düğümde makul olmayan uzun bir hesaplama süresi gerektirecek veri boyutunda problemleri çözmektir. Diğer bir neden, mevcut bir MPI uygulamasını GPU'larla hızlandırmak veya mevcut tek düğümlü çoklu GPU uygulamasının birden fazla düğüm arasında ölçeklendirilmesini sağlamaktır. MPI ve CUDA birleştirildiğinde, ana bilgisayar arabellekleri yerine GPU arabelleklerine gönderilmesi gerekmektedir [14].

2.6.3 CUDA Rastgele Sayı Kütüphanesi (CUDA Random Number Generation Library-cuRAND)

cuRAND kütüphanesi, yüksek kaliteli sözde rastgele ve yarı rastgele sayıların basit ve verimli üretimine odaklanan olanaklar sağlamaktadır. Sahte rastgele bir sayı dizisi, gerçekten rastgele bir dizinin istatistiksel özelliklerinin çoğunu karşılamaktadır. Ancak deterministik bir algoritma tarafından üretilir. N-boyutlu bir alanı eşit olarak doldurmak için tasarlanmış deterministik bir algoritma tarafından n-boyutlu noktaların yarı-rastgele bir dizisi oluşturulmaktadır.

cuRAND iki parçadan oluşur; CPU tarafında bir kütüphane ve GPU başlık dosyası (header file). CPU tarafındaki kütüphane, diğer herhangi bir CPU kitaplığı gibi ele alınır. Kullanıcılar, işlev bildirimlerini almak için "/include/curand.h" başlık dosyasını ekler ve ardından kütüphaneye bağlanır. GPU veya CPU'da rastgele sayılar oluşturulabilir. GPU'da üretim için, kütüphaneye yapılan çağrılar CPU'da gerçekleştirilir, ancak rastgele sayı oluşturmanın asıl işi GPU'da gerçekleştirilmektedir. Ortaya çıkan rastgele sayılar, GPU'nun global belleğinde saklanmaktadır. Kullanıcılar daha sonra rastgele sayıları kullanmak için kendi Kernel'larını çağırabilirler veya daha fazla işlem için rastgele sayıları CPU'ya geri kopyalayabilirler.

cuRAND'ın ikinci parçası, "/include/curand_Kernel.h" GPU başlık dosyasıdır. Bu dosya, rastgele sayı üreteci durumlarını ayarlamak ve rastgele sayı dizileri oluşturmak için GPU işlevlerini tanımlar. Kullanıcı kodu bu başlık dosyasını içerebilir ve kullanıcı tarafından yazılan Kernel'lar daha sonra başlık dosyasında tanımlanan cihaz işlevlerini çağırabilir. Bu yapı, rastgele sayıların global belleğe yazılmasına ve daha sonra okunmasına gerek kalmadan rastgele sayıların üretilmesine ve kullanıcı Kernel'ları tarafından hemen tüketilmesine izin vermektedir [15].

3. OPTİMİZASYON VE CUDA ÇÖZÜMLERİ

Bu bölümde optimizasyon kavramı açıklanacak, metasezgisel optimizasyon algoritmalarından ve CUDA paralel programlama yaklaşımı ile yapılan çalışmalardan bahsedilecektir.

3.1 Optimizasyon

Optimizasyon, bir problemin tüm çözümleri içerisinden en iyinin ortaya çıkarılması şeklinde tanımlanır [16]. Ancak bazı problemler için kabul edilebilir bir maliyet ile çözüme ulaşmak mümkün olmayabilir. Bu durumda sezgisel olarak geliştirilen ve doğal hayatı taklit ederek problem çözümüne ulaşabilen metasezgisel algoritmalar kullanılmaktadır [17].

Bir optimizasyon problemi doğru bir şekilde formüle edildikten sonra ana görev, doğru matematiksel teknikleri kullanarak bazı çözüm prosedürleriyle en uygun çözümleri bulmaktır. Mecazi anlamda en iyi çözümü aramaya şöyle bir örnek verilebilmektedir. Bir bölgede hazine arandığını düşünelim. En olası senaryo, bazı ipuçları ararken rastgele bir yürüyüş yapacağımızdır. Bir yere neredeyse rastgele bakarız, sonra başka bir makul yere, sonra başka bir yere gidebiliriz. Bu tür rastgele yürüyüş modern arama algoritmalarının temel özelliğidir. Tüm izlenen yollar yörünge tabanlı bir aramadır. Alternatif olarak bir gruptan hazine avına çıkmasını ve grubun her bireyinin bilgi paylaşmasını isteyebiliriz. Bu şekilde bilgi paylaşılan yerlere bireyler tekrar tekrar bakmaz. Arama işlemi uzun süre alacak fakat hazine değerli ise bu yöntemle nihai hazineyi bir grupla bulmak mümkün olacaktır. Arama bölgesini daha da daraltmak için tecrübeli avcılar tutabilir ve yenileri gruba eklenebilir. Hemen hemen tüm modern metasezgisel algoritmalarda grubun her bir üyesinin yetkinliği (fitness) ve sistemin geçmiş bilgisi (aranan bölgelerin bir yerde tutulması-use of memory) önem arz etmektedir. Bu şekilde daha verimli metasezgisel algoritmalar tasarlanmaktadır. Bazı metasezgisel algoritmalara arıların polen için uygun çiçeği bulması ve polen alınan çiçeklerin tekrar aranmaması, penguenlerin sürü halinde balık gruplarını avlamaları örnek olarak verilebilmektedir [17] .

Geleneksel veya klasik algoritmaların çoğu deterministir. Örneğin, doğrusal programlamada simpleks yöntemi belirleyicidir. Bazı deterministik optimizasyon algoritmaları gradient bilgisini kullanır, bunlara gradient tabanlı algoritmalar denir. Örneğin, iyi bilinen Newton-Raphson algoritması, işlev değerlerini ve türevlerini kullandığından ve düzgün tekli

problemler için son derece iyi çalıştığı için gradient tabanlıdır [18]. Bununla birlikte, nesnel işlevde bir süreksizlik varsa, Newton-Rapshon gibi algoritmalar iyi çalışmaz. Bu durumda, gradyan olmayan bir algoritma tercih edilir.

Gradyan tabanlı veya gradyan içermeyen algoritmalar herhangi bir türev değil, sadece fonksiyon değerlerini kullanır. Hooke-Jeeves [19] örüntü arama ve Nelder-Mead downhill simpleks [20], gradient içermeyen algoritmalara örnektir.

Stokastik algoritmaların genel olarak iki tipi vardır. Bunlar sezgisel ve metasezgisel algoritmalardır. Sezgiselin anlamı, bulmak için deneme yanılma yoluyla keşfetmektir. Zor bir optimizasyon problemine kaliteli bir çözüm makul bir sürede bulunabilir. Fakat en iyi çözüme ulaşıldığının garantisi yoktur. Sezgisel algoritmaların geliştirilmesiyle metasezgisel algoritmalar ortaya çıkmıştır[21]. Burada meta "ötesinde” veya “daha yüksek düzey” anlamına gelir ve genellikle basit sezgisel yöntemlerden daha iyi performans gösterirler.

Son yıllarda yayınlanan makaleler, rastgelelik ve yerel arama içeren tüm stokastik algoritmaları metasezgisel olarak adlandırmaya eğilimlidir. Rastgeleleştirme, yerel aramadan küresel ölçekte aramaya geçmek için iyi bir yol sağlar. Bu nedenle, tüm metasezgisel algoritmalar en iyi çözüme ulaşmak için uygun yapıda olmayı amaçlamaktadır. Sezgisel algoritmaların temel amacı problemin en kısa sürede en verimli şekilde çözümlenmesidir. Bir metasezgisel algoritmanın çeşitlendirme ve yoğunlaştırma olmak üzere iki ana bileşeni vardır. Çeşitlendirme, global ölçekte arama alanını keşfetmek için çeşitli çözümler üretmek anlamına gelirken yoğunlaştırma, bu bölgede mevcut iyi bir çözümün bulunduğu bilgisini kullanarak yerel bir bölgede aramaya odaklanmak anlamına gelmektedir. En iyinin seçimi, çözümlerin en iyiye yaklaşmasını sağlarken, rastgeleleştirme yoluyla çeşitlendirme, çözümlerin yerel en iyide sıkışmasını önler ve aynı zamanda çözümlerin çeşitliliğini arttırır. Bu iki ana bileşenin iyi birleşimi genellikle global en iyilerin elde edilebilir olmasını sağlamaktadır [21].

Her yerde bulunmasına rağmen, problem çözme için bilimsel bir yöntem olarak sezgisel algoritmalar modern bir olgudur, ancak sezgisel yöntemin ilk kez ne zaman kullanıldığını belirlemek zordur. Alan Turing, muhtemelen İkinci Dünya Savaşı sırasında sezgisel algoritmaları kullanan ilk kişiydi. Alan Turing sezgisel arama yaklaşımı ile Almanların

Enigma şifreleyicinin mesajlarını çözebilmiştir. Alan Turing'in bu çalışması makine öğrenmesi, sinir ağları ve evrimsel algoritmalara öncü olmuştur [22].

1960 ve 1970'ler genetik algoritmalar için başlangıç süreci olmuştur. Genetik algoritmalar ilk olarak John Holland tarafından geliştirilmiştir [23]. 1963 senesinde Berlin Teknik Üniversitesi'nde Ingo Rechenberg ve Hans Paul Schwefel tarafından havacılık uzay mühendisliğinde kullanılmıştır [24]. Marco Dorigo ve arkadaşları Stanford Üniversitesi'nde doktora tezi çalışmasını doğal algoritmalar ve karınca koloni üzerine yenilikçi bir çalışma ile tamamlamıştır [25]. Daha sonraki yıllarda Zong Woo harmonik arama çalışması ile geniş çaplı optimizasyon problemlerine çözüm üretmiştir [26]. Nakrani bal arısı algoritması [27], Karaboğa yapay arı kolonisi algoritması üzerine çalışmalar yapmıştır [28].

3.2 CUDA Kullanılarak Yapılan Metasezgisel Algoritma Çözümleri

Gün geçtikçe CUDA paralel uygulamaları ve kaynak gücünün artışı, metasezgisel algoritmalarda paralel uygulama sayısını arttırmakta ve literatüre yeni çalışmaların eklenmesini sağlamaktadır. Bu bağlamda yapılan literatür taramaları aşağıda belirtilmiştir.

Mussi ve Cagnoni çalışmalarında bir metasezgisel algoritma olan PSO algoritmasına CUDA paralel programlama yaklaşımı uygulamışlardır. Bu çalışmada yapılan testlerde seri programlamaya göre 22 kat daha fazla hız gözlemlemişlerdir [29]. Debanjan Datta vd. PSO algoritmasının bir CUDA versiyonunu uygulamışlardır. Bunu öz potansiyel, manyetik ve direnç verilerini tersine çevirmek için kullanmışlardır. Algoritmanın CUDA versiyonu, aynısının verimli bir CPU uygulamasıyla karşılaştırmışlardır. CUDA sürümünün sonuçlarının CPU sürümünden iyi olduğunu belirtmişlerdir [30]. Solomon vd. çalışmalarında PSO'yu heterojen dağıtılmış bir bilgi işlem ortamında görev eşleştirme probleminde kullanmışlardır. Bu problem çok boyutlu olduğu için paralel yaklaşımda CUDA tercih edilmiştir. Paralel programlama yaklaşımının seri programlama yaklaşımına göre bu problemi çözmede 37 kat hızlanma sağladığı tespit edilmiştir [31]. Platos vd. genellikle büyük zaman karmaşıklığına sahip belge sınıflandırma probleminde paralel CUDA ile PSO tabanlı döküman sınıflandırma algoritmasını kullanmışlar ve seri programlamaya göre hızlanma tespit etmişlerdir [32]. Luo vd. arama problemlerinde optimale yakın çözüm bulmak için doğal bal arılarının davranışlarından ilham alan arı algoritması için GPU üzerinde çalışan paralel bir yaklaşımla uygulama geliştirmişler ve bu uygulamayı CUBA (CUDA based Bees Algorithm) olarak adlandırmışlardır. CUDA ile

yapılan bu paralel yaklaşımda test fonksiyonunun türüne bağımlı olarak 13-56 kat arasında hızlanma gözlenmiştir [33]. Janousešek vd. yapay arı kolonisi ve bal arısı sürüsünün davranışına dayanan metasezgisel bir optimizasyon algoritması üzerinde çalışmışlardır. Bu çalışmada arıların, diğer arılardan büyük ölçüde bağımsız olmasının algoritmayı paralel uygulamaya uygun hale getirdiğinden bahsedilmiştir. Bu çalışmada, algoritmanın kendisi ve CUDA kullanarak paralel olarak geliştirilmesi anlatılmıştır. Çalışma zamanı hızlandırması, optimizasyon için yaygın olarak kullanılan birkaç test işlevinde gösterilmiştir. Bu geliştirilen paralel yapay arı kolonisi yaklaşımı gerçek verileri sınıflandırma probleminde kullanılmıştır. Geliştirilen bu sınıflandırma probleminde 86,25 kat hızlanma tespit etmişlerdir [34]. Wang vd. bilimsel iş akışı çizelgeleme uygulamalarında binlerce görev düğümü olduğunu belirtmişler ve bu sürecin büyük bir işlem yüküne sahipliğinden bahsetmişlerdir. Bilimsel akış planlaması için paralel çalışacak karınca koloni algoritmasını önermişlerdir. Paralel geliştirme aşamasında CUDA ve C dilini kullanmışlardır. 1000 görev düğümü için uygulamalarının 5 saniyede çalıştığını gözlemlemişlerdir. Seri programlama yaklaşımına göre 20,7 kat hızlanma olduğunu belirtmişlerdir [35]. Kai vd. çalışmasında CUDA'ya dayalı grafik renklendirme problemini çözmek için yeni bir paralel genetik algoritma sunmuşlardır. Başlatma, çaprazlama, mutasyon ve seçim operatörleri iş parçacıklarında paralel olarak tasarlamışlardır. Ayrıca, algoritmalarının performansı, standart DIMACS kıyaslama grafikleri kullanılarak diğer grafik renklendirme yöntemleriyle karşılaştırmışlar ve karşılaştırma sonucu, algoritmalarının hesaplama süresi ve grafik örnekleri boyutu ile daha rekabetçi olduğunu göstermişlerdir [36]. Silva ve Bastos, çalışmalarında platformlarının grafik işlem birimlerinde bulunan ve paylaşılan belleği kullanan PSO algoritması için verimli bir uygulama önermişlerdir. Problemdeki sürüyü alt sürülere ayırarak paralel geliştirme yapmışlardır. Ayrıca alt sürünün GPU global belleğini kullanarak bilgi alışverişinde bulunmasına ve iş birliği yapmasına izin vermek için iki iletişim mekanizması ve iki topoloji önermişlerdir. Her biri 32 parçacık ve 32 boyuta sahip 8 alt sürünün sonuçları, CUDA için seri uygulama ve PSO başlangıç uygulaması ile karşılaştırıldığında 100 kata kadar hızlanma olduğunu göstermişlerdir [37]. Zarrabi vd. çalışmasında yer çekim arama algoritmasının çalışma yürütme süresini azaltmak için CUDA mimarisi tabanlı paralel hesaplama yaklaşımı önermişlerdir. Paralel algoritmaların hesaplama verimliliği ve GPU mimarisinden etkin bir şekilde yararlanma yeteneği hakkında derinlemesine bir çalışma gerçekleştirmişlerdir. Ek olarak, bir dizi standart kıyaslama optimizasyon işlevi üzerinde paralel ve sıralı yerçekimi arama algoritmasının karşılaştırmalı bir çalışmasını yapmışladır. Sonuçlarında, karmaşık ve hesaplama açısından yoğun paralel uygulamalar için CUDA

tabanlı uygulamanın faydasını vurgulayan önemli bir hızlanma göstermişlerdir. Bazı test fonksiyonlarında bu hızlanma 3 ila 28 kat arasında gerçekleşmiştir [38]. Ouyang vd. çalışmalarında bir boyutlu ısı iletim denklemini lineer denklemler sistemine dönüştürmüşler ve paralel hibrit PSO algoritması kullanarak çözmüşlerdir. Paralel yaklaşımda CUDA mimarisini temel almışlardır. Paralel hibrit parçacık sürü optimizasyon yaklaşımının standart sapma ve hızlanma açısından rekabetçi olduğunu öne sürmüşlerdir [39]. Bukata vd. çalışmalarında, kaynak kısıtlı proje çizelgeleme problemini çözmek için tabu arama algoritmasının paralel geliştirmesini önermişlerdir. Paralel yaklaşımda CUDA mimarisini tercih etmişlerdir. Paralel uygulama yaptıkları bu çalışmada tabu arama algoritmasının çözüm kalitesi ve saniye başına değerlendirilen çizelge sayısı açısından mevcut diğer tabu arama yaklaşımlarından daha iyi performans gösterdiğini öne sürmüşlerdir. CPU da yapılan işleme göre 10,5 ila 42,7 kat arasında hızlanma tespit etmişlerdir [40]. Kıran ve Çınar ağaç-tohum algoritmasının CUDA mimarisinde paralel uygulaması üzerine çalışma yapmışlardır. Problem boyutunu 10 olarak alınmış ve farklı popülasyon ve CUDA Block sayıları için hızlanma analizi yapılmıştır. Deneysel çalışmalarında paralel algoritmanın seri yönteme göre 184,65 kata kadar hızlanma sağladığını göstermişlerdir [16].

Literatür taramasından da anlaşılacağı üzere CUDA ile paralel olarak geliştirilmiş metasezgisel algoritmaların CPU'da çalıştırılan algoritmalara göre hızlı olduğu görülmektedir.

4. UYGULAMA VE DENEYSEL ÇALIŞMALAR

Bu bölümde geliştirilen uygulama ve deneysel çalışmalar ayrıntılı bir şekilde anlatılmaktadır.

4.1 Parçacık Sürü Optimizasyon Algoritması

PSO, sürü olarak dolaşan kuşlardan ve balıklardan gözlem yapılarak, Kennedy ve Eberhart tarafından ortaya konulmuş bir metasezgisel algoritmadır [41]. Yiyecek ve güvenlik ihtiyacının bulunduğu sürülerde, her bir sürüye ait bireyin sergilediği rastgele hareketlerin sürünün hedeflerine kolay varmasına yol açtığı görülmektedir. PSO, sürünün bireyleri içerisinde yiyecek ve güvenli alan gibi bilgi paylaşımını ve iletişimi baz almaktadır. Arama işlemi belirlenen durma kriteri sağlanıncaya kadar yapılır [6].

Sürünün her bir elemanı parçacık olarak tanımlanmaktadır. Parçacıklardan oluşan grup veya toplu popülasyona da sürü denmektedir. Her bir birey (parçacık) kendisinin konumunu, daha önceki deneyimlerini dikkate alarak kendi bulunduğu pozisyonu olabildiğince iyi konuma yöneltir. PSO'nun yapısı sürü de bulunan parçacıkların diğer parçacıkların tecrübesinden yararlanarak iyi bir konuma yakınsamasına dayanmaktadır. Yakınsama hızı sürekli değişken bir durumdur ve parçacıklar genellikle bir önceki pozisyonundan daha iyi bir pozisyona erişirler. Amaca yani en iyi konuma ulaşıncaya kadar böyle devam etmektedir.

Kısaca bir örnekle açıklamak gerekirse; bir ormanda bir kuş sürüsü ve bir yiyecek noktası düşünebilir. Bu sürüdeki her kuş (parçacık) ulaşılacak yiyecek noktasını bilmemektedir. Yiyecek arama sürecinde kuşlar ormanda uçarken hızlarını korumaktadırlar. Bu süreçte her kuş diğerlerinden daha iyi olduğunu kanıtlamayı hedeflemektedir ve sezgilerine (bilişsel bileşen) dayalı olarak yiyecek bulmaya çalışmaktadır. Ancak diğerlerini (sosyal bileşen) taklit etme eğiliminde olduğu için, grubunun deneyim ve bilgisinden de etkilenmektedirler.

Aşağıdaki sözde kod ve denklemler PSO algoritmasını çalışma şeklini temsil etmektedir [6].

```
For herbir parçacık
    Rastgele başlangıç pozisyonları ve hızları ile sürü oluştur
End
Do
    For herbir parçacık
        Uygunluk değeri hesapla
        If uygunluk değeri en iyi uygunluk değerinden daha iyi ise
             Mevcut uygunluk değerinin yeni uygunluk değeri ile güncelle
    End
    Mevcut iterasyondaki yerel en iyiler arasından küresel en iyi seç
    For herbir parçacık
        Parçacıkların hızlarını hesapla (4.1)
        Parçacıkların konumunu güncelle (4.2)
    End
While maksimum iterasyon veya hata kriterine ulaşıncaya kadar devam et
```

Parçacık hız hesaplama ve konum güncelleme denklemleri sırasıyla Denklem 4.1 ve Denklem 4.2'de verilmiştir [6];

$$V_i^{t+1} = \omega V_i^t + c_1 r_1 (X_{best(i)}^t - X_i^t) + c_2 r_2 (X_{bestglobal}^t - X_i^t) \tag{4.1}$$

$$X_i^{t+1} = X_i^t + V_i^{t+1} \tag{4.2}$$

Bu denklemlerde, ω: Atalet değerini, V_i: i. parçacığın hızını, $c_{1,} c_2$: hızlanma katsayısını, X_i: i. parçacığın o andaki konumu, $X_{best(i)}$: i. parçacığın o ana kadarki en iyi konumu, $X_{bestglobal}$: sürüde o ana kadar elde edilen en iyi konum ve $r_{1,} r_2$: 0 ve 1 arasında rastgele üretilen sayıları temsil etmektedir.

4.2 Test Fonksiyonları

Tablo 4.1'de tek modlu Tablo 4.2'de çok modlu test fonksiyonları ve formülleri verilmiştir. Bu test fonksiyonları uygulamada kullanılmıştır [42], [43]. Minimum değer, fonksiyonun sınırlar çerçevesinde ulaşabileceği en küçük değeri temsil etmektedir.

Tablo 4.1: Tek modlu test fonksiyonları.

Fonksiyon Numarası	Fonksiyon Adı	Fonksiyon Formülü	Sınırlar	Minimum Değer
f_1	Rosenbrock	$f(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i{}^2)^2 + (x_i - 1)^2]$	$x_i \epsilon$ $[-2.048, 2.048]$	0
f_2	Sphere	$f(x) = \sum_{i=1}^{d} x_i{}^2$	$x_i \epsilon$ $[-5.12, 5.12]$	0
f_3	Schwefel 2.22	$f(x) = \sum_{i=1}^{d} \|x_i\| + \prod_{i=0}^{d} \|x_i\|$	$x_i \epsilon$ $[-100, 100]$	0
f_4	Maximization	$f(x) = max\{\|x_i\|, 1 \leq i \leq d\}$	$x_i \epsilon$ $[-100, 100]$	0
f_5	Step	$f(x) = \sum_{i=1}^{d} ([x_i + 0.5])^2$	$x_i \epsilon$ $[-100, 100]$	0
f_6	Quartic Noise	$f(x) = \sum_{i=1}^{d} ix_i^4 + random[0,1)$	$x_i \epsilon$ $[-1.28, 1.28]$	0
f_7	Griewank	$f(x) = \frac{1}{4000} \sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$x_i \epsilon$ $[-600, 600]$	0
f_8	Schwefel 2.20	$f(x) = \sum_{i=1}^{d} \|x_i\|$	$x_i \epsilon$ $[-100, 100]$	0
f_9	Schwefel 2.23	$f(x) = \sum_{i=1}^{d} x_i^{10}$	$x_i \epsilon$ $[-10, 10]$	0
f_{10}	Sum Squares	$f(x) = \sum_{i=1}^{d} ix_i^2$	$x_i \epsilon$ $[-10, 10]$	0
f_{11}	Exponential	$f(x) = -exp(-0.5 \sum_{i=1}^{d} x_i^2)$	$x_i \epsilon$ $[-1, 1]$	-1
f_{12}	Powell	$f(x) = \sum_{i=1}^{d} \|x_i\|^{i+1}$	$x_i \epsilon$ $[-1, 1]$	0

Tablo 4.2: Çok modlu test fonksiyonları.

Fonksiyon Numarası	Fonksiyon Adı	Fonksiyon Formülü	Sınırlar	Minimum Değer
f_{13}	Periodic	$f(x) = 1 + \sum_{i=1}^{d} sin^2(x_i) - 0.1exp(-\sum_{i=1}^{d} x_i^2)$	$x_i \epsilon\ [-10,10]$	0.9
f_{14}	Salomon N.1	$f(x) = 1 - (\cos(2\pi \sqrt{\sum_{i=1}^{d} x_i^2}) + 0.1 \sqrt{\sum_{i=1}^{d} x_i^2}$	$x_i \epsilon\ [-100,100]$	0
f_{15}	Xin-She Yang N.2	$f(x) = (\sum_{i=1}^{d} \lvert x_i \rvert) exp(-\sum_{i=1}^{d} \sin(x_i^2))$	$x_i \epsilon\ [-2\pi, 2\pi]$	0
f_{16}	Alphine N.1	$f(x) = \sum_{i=1}^{d} \lvert x_i \sin x_i + 0.1 x_i \rvert$	$x_i \epsilon\ [0,10]$	0
f_{17}	Ackley	$f(x) = -20exp(-0.2 \sqrt{\frac{1}{n}\sum_{i=1}^{d} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{d} \cos(2\pi x_i)) + 20 + e$	$x_i \epsilon\ [-32,32]$	0
f_{18}	Rastrigin	$f(x) = 10d + \sum_{i=1}^{d} [{x_i}^2 - 10\cos(2\pi x_i)]$	$x_i \epsilon\ [-5.12,5.12]$	0

4.3 Geliştirilen Uygulama

Bu çalışmada, PSO algoritması üç yöntem ile uygulanmıştır. Birinci yöntem algoritmanın CPU'da seri olarak uygulanmasıdır. İkinci yöntem GPU'da paralel uygulanmasıdır. Üçüncü yöntem ise hem seri hem de paralel yaklaşım içeren hibrit modeldir [44]. Her deneyin dinamik bir yapıda gerçekleştirilebilmesi amacıyla test fonksiyonları dinamik yapıda kodlanmıştır.

Uygulama aşamasında hibrit ve CPU'da çalışacak kod parçaları aynı program düzeninde geliştirilmiştir [44]. Bunun en büyük sebebi hibrit yapıda, test fonksiyonları uygulanırken ve global en iyi hesaplanırken seri programlama yöntemi kullanılmasıdır. GPU'da çalışacak kod parçaları ise farklı bir program düzeninde yazılmıştır.

CUDA kullanılarak geliştirilen uygulamada test fonksiyonlarının her biri paralel yapıda uygunluk değeri hesaplayacak şekilde kodlanmıştır. Fakat CUDA paralel uygulamasında ve hibrit uygulamada her parçacık için konum ve hız değerleri güncellenirken Kernel'lar oluşturulmuş ve bu işlemler paralel programlama mantığında oluşturulan Thread ve Block'larla yapılmıştır.

Her üç farklı uygulama geliştirilirken, çalışma yöntemlerinin ve hızlarının programlama dilinin işlevlerinden etkilenmemesi için CUDA'nın geliştirilme dili olan C programlama dili tercih edilmiştir. Geliştirmeler aşamasında işletim sistemi olarak açık kaynak kodlu Debian tabanlı Ubuntu 20.04 işletim sistemi tercih edilmiştir. Bu tarz işletim sistemlerinde arkada çalışan sistem uygulaması daha az olduğu için CPU'daki yük daha az olmaktadır. Bu sayede uygulamanın bağımsız çalıştırmalarında süre daha verimli şekilde hesaplanabilmektedir. Uygulama geliştirme ve test fonksiyonları ile yapılan deneylerde Tablo 4.3'te bulunan özelliklerdeki günlük kullanıma uygun dizüstü bilgisayar kullanılmıştır.

Tablo 4.3: Kullanılan donanım bilgileri.

	CPU	**GPU**
İşlemci	Intel (R) Core (TM) i7-7700HQ CPU	Nvidia GeForce 1050 Ti
Çekirdek Sayısı	4 Çekirdek 8 Thread	6 Multiprocessors 128 CUDA cores: 768 CUDA Cores
İşlemci Saat Hızı	2.80GHZ	1.62 GHZ
Bellek Boyutu	24 GB	4 GB
Bellek	DDR4	DDR5
CUDA Versiyonu	-	11.3

4.3.1 PSO CPU Uygulaması

Bu çalışmada ilk olarak standart CPU'da seri olarak çalışan PSO geliştirilmiştir. Tüm geliştirme aşamasında her bir rastgele sayı üretme işlemi CPU üzerinde yürütülmüştür. Ayrıca belirlenen ve bölüm 4.2'de bahsedilen her bir test fonksiyonu çalıştırılarak deney çıktıları üretilmiştir. Uygulama tamamen C programlama dilinde geliştirilmiştir. CPU'da geliştirilen uygulamada, tüm bellekten veri ayırma işlemleri işlemcinin direkt eriştiği rastgele erişimli bellek (Random Access Memory- RAM) üzerinde yapılmıştır.

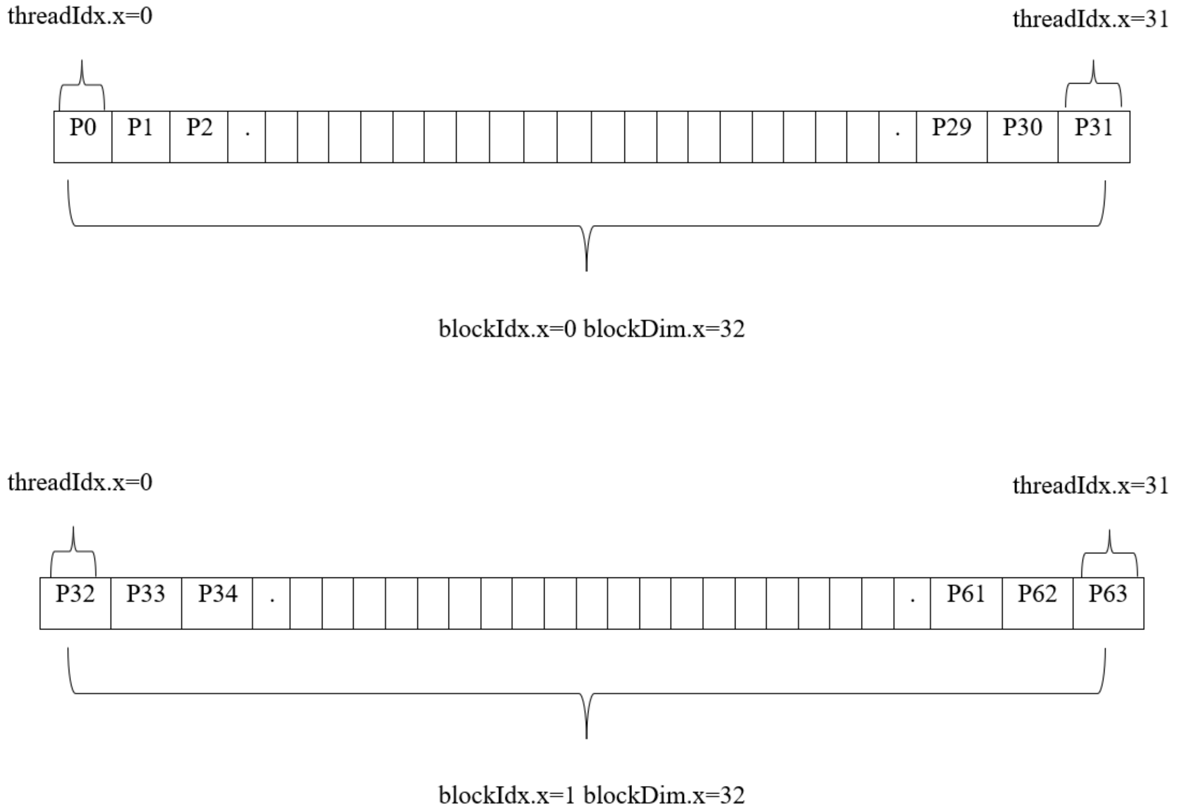
Algoritma geliştirme aşamasında parçacıkların ilk hız ve konumlarının belirlenmesinde, lokal minimumlarının hesaplanmasında, hız ve konum güncellemelerinde ve global en iyinin belirlenmesinde tamamen seri çalışacak yapı tercih edilmiştir. Ayrıca bu yapı geliştirilirken kullanılan rastgele sayı üreticiler ve ilk konum belirleme yaklaşımları aşağıda anlatılan hibrit yapıda da kullanılmıştır. Uygulama geliştirme aşamaları Şekil 4.1'de bulunan akış diyagramında mevcuttur.

Şekil 4.1: PSO CPU uygulaması akış diyagramı.

4.3.2 PSO Hibrit Uygulaması

Hibrit yapıda geliştirilen uygulamada rastgele sayı üretimi, sürünün her parçacığının hız ve konum bilgilerinin oluşturulması için CPU kullanılır. Bu veriler “cudaMemcpy” (host →device) ile GPU belleğine aktarılır. Uygunluk değerleri GPU’da seri programlama yaklaşımı ile hesaplanır. Parçacığın o ana kadarki kendi en iyisinin bulunmasında paralel, global en iyinin bulunmasında seri programlama yöntemi kullanılmıştır. Burada amaçlanan, veri kopyalama işlemi ve seri programlama ile bulunan uygunluk değeri ve global en iyinin çalışma zamanına etkisini gözlemlemektir.

CUDA tarafında, algoritmada parçacık sayısının 64 olmasından dolayı konum, hız ve lokal minimum güncellenirken 2 Block’lu ve 64 Thread’li yapı kullanılmıştır. Block ve Thread yerleştirmesi Şekil 4.2’de gösterildiği gibidir. Bu şekilde her bir parçacık için bir Thread atanmıştır.



Şekil 4.2: Hibrit yaklaşım Thread Block yapısı.

Her bir parçacığa Thread atandığında her bir Thread’in erişebileceği boyut×parçacık sayısı kadar konum ve hız için bellekten yer ayrılmıştır. Konum ve hız bilgisi yeniden

hesaplandığında Thread'ler kendilerine ait indeksler üzerinden bu bellek alanlarına ulaşarak bu alanlardaki değerleri güncellemektedirler.

Aşağıdaki örnek kod parçalarında GPU'da hız ve pozisyonlar için bellek ayrılmasında kullanılan "cudaMalloc" metotları gösterilmektedir.

```
cudaMalloc((void**)&pozisyon,parcacik_sayisi * problem_boyutu * sizeof(float))

cudaMalloc((void**)&hiz,parcacik_sayisi * problem_boyutu * sizeof(float))
```

Aşağıdaki örnek kod parçalarında ise CPU'da oluşturulan ilk konumların GPU tarafına kopyalanmasında kullanılan "cudaMemcpy" metotları gösterilmektedir.

```
cudaMemcpy(pozisyon_CPU,pozisyon,sizeof(float)*parcacık_sayisi *
problem_boyutu,CUDAMemcpyHostToDevice)

cudaMemcpy(hiz_CPU,hiz,sizeof(float)*parcacık_sayisi *
problem_boyutu,CUDAMemcpyHostToDevice)
```

Şekil 4.3'te 128 boyutlu bir parçacık için bellekte oluşturulan yapı temsili olarak gösterilmektedir. Her parçacık için bahsedildiği üzere pozisyon ve hız bu şekilde GPU belleğinde yer almaktadır.

P0

D0	D1	D2	.																			.	D125	D126	D127

Şekil 4.3: Her bir parçacık için tutulan 128 boyutlu belleğin temsili.

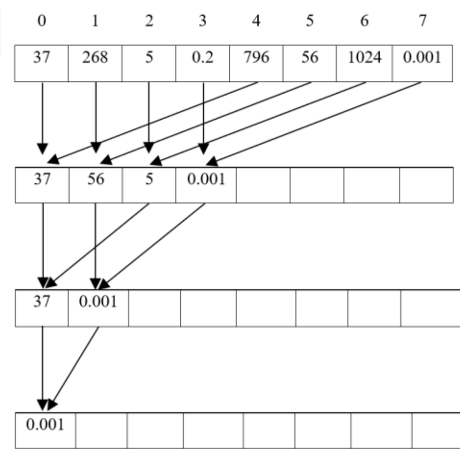
Şekil 4.4'te PSO hibrit uygulamasının akış diyagramı verilmektedir. Bu diyagramda CPU'da yapılan işlemler mavi, GPU'da yapılan işlemler ise yeşil renk ile gösterilmiştir.

Şekil 4.4: PSO hibrit uygulaması akış diyagramı.

4.3.3 PSO CUDA Uygulaması

Uygulama geliştirme aşamasında uygunluk değeri hesaplamalarında, hız ve konum güncelleştirilmelerinde CUDA üzerinde paralel programlama yaklaşımı uygulanmıştır. Hibrit yöntemde zaman kayıplarından biri CPU'dan GPU global belleğine "cudaMemcpy" aracılığı ile veri kopyalamasından kaynaklanmaktadır. Bu sorunu ortadan kaldırmak amacıyla rastgele sayı üretimi için CUDA tabanlı CURAND kütüphanesi kullanılmıştır. Thread ve Block yerleştirme düzeni, hibrit yöntem ile aynı olması için Şekil 4.2'de gösterildiği gibi her bir Block'ta 32 Thread olacak şekilde 2 Block'tan oluşturulmuştur.

Ayrıca bu yöntemde global en iyi sonuç, CUDA Reduction kullanılarak bulunmuştur. Reduction yaklaşımı paralel olarak bellekte bulunan verilerin karşılaştırılarak her seferinde minimumlarının veya maksimumların bulunması, toplamların veya çarpımların hesaplanması gibi operasyonlardan geçirilmesiyle bir alt belleğe indirgenmesi olarak tanımlanabilmektedir. Buradaki indirgeme işlemleri birbirleri ile Block içerisinde haberleşen Thread'ler tarafından yürütülmektedir. Şekil 4.5'te örnek bir Reduction işlemi gösterilmektedir.



Şekil 4.5: Örnek paralel CUDA Reduction yapısı.

CUDA kullanılarak geliştirilen uygulamanın akış diyagramı Şekil 4.6'da sunulmaktadır. Bu akış diyagramında mavi renk ile gösterilen işlemler CPU'da, yeşil renk ile gösterilen işlemler ise GPU'da yapılmıştır.

Şekil 4.6: PSO CUDA paralel uygulaması akış diyagramı.

4.4 Test Sonuçları

Çalışmada CPU'da seri, CPU-CUDA hibrit ve sadece CUDA'da paralel çalışan PSO algoritması, 18 farklı test fonksiyonu için uygulanmıştır. 100'er bağımsız denemeden elde edilen sonuçlardan ortalama, en iyi ve en kötü olanlar verilmiştir. Ayrıca algoritmanın kararlı çalışmasını incelemek için amacıyla standart sapma değeri sunulmuştur. Bunun yanında her bir test fonksiyonuyla yapılan bağımsız denemeler için harcanan sürelerin ortalaması verilmiştir. Bu süreler kullanılarak hibrit ve CUDA yöntemleri için hızlanma hesaplanmıştır. Hızlanma verileri, seri yöntem için harcanan sürenin paralel yöntemde harcanan süreye bölümüyle elde edilmiştir. Tüm uygulamalar, parçacık sayısı 64, problem boyutu 64 ve 128, iterasyon sayısı 1000, 2500 ve 5000 için yapılmıştır. PSO algoritmasında w değeri 0,72984 [45] olarak kullanılmıştır.

Deneysel çalışmaların yapılma aşamasında, işletim sistemi üzerinde arkada çalışan bir uygulama olmamasına dikkat edilmiştir. Ayrıca GPU'da herhangi bir etki yaratmaması amacıyla kullanıcı arayüz ekranı CPU üzerinde bulunan gömülü GPU üzerine yönlendirilmiştir.

Tablolarda metot olarak belirtilen kısımda "CUDA" paralel yöntemi, "HİBRİT" işlerin CPU ve CUDA arasında paylaştırıldığı, "CPU" ise seri programlama yaklaşımını ifade etmektedir.

Tablo 4.4'te 64 boyutlu problemlerin 1000 iterasyondaki sonuçları verilmektedir. En yüksek hızlanmalar CUDA için f_3 fonksiyonunda 6,359721, hibrit için f_3 fonksiyonunda 3,116508 olarak elde edilmiştir. Ortalama hızlanma CUDA için 3,429504, hibrit için 1,713861 olarak bulunmuştur. En düşük hızlanma değerleri ise CUDA için f_{12} fonksiyonunda 2,174073 ve hibrit için f_1 fonksiyonunda 1,175112 olarak elde edilmiştir. Tablo 4.5, 128 boyutlu problemlerin 1000 iterasyondaki sonuçlarını vermektedir. En yüksek hızlanmalar CUDA için f_3 fonksiyonunda 7,034139, hibrit için f_3 fonksiyonunda 7,136033 olarak hesaplanmıştır. En düşük hızlanmalara bakıldığında ise CUDA için f_1 fonksiyonunda 2,167305 ve hibrit için f_9 fonksiyonunda 1,059133 bulunmuştur. Ortalama hızlanma CUDA için 3,460479, hibrit için 2,235399 olmuştur.

Tablo 4.6 ve Tablo 4.7'de sırasıyla 64 ve 128 boyutlu problemlerin 2500 iterasyondaki sonuçları verilmektedir. Her iki yöntem için en yüksek hızlanmalar f_3 fonksiyonunda, en düşük hızlanmalar f_1 fonksiyonunda bulunmuştur. Problem boyutu 64 olduğunda en yüksek hızlanma değerleri, CUDA için 7,619271, hibrit için 3,465646, en düşük hızlanma değerleri ise CUDA için 1,942267 ve hibrit için 1,087825 olarak elde edilmiştir. Ortalamaya bakıldığında ise CUDA 3,382932, hibrit 1,731462 kat hızlanma üretmiştir. Problem boyutu 128 için yapılan testlerde ise CUDA 6,976619, hibrit 7,012002 kat kadar hızlanma sağlamıştır. 128 boyutlu problemlerin 2500 iterasyonda çözüldüğü bu test, hibrit yöntemin CUDA'ya göre daha yüksek hızlanma ürettiği tek durumdur. En düşük hızlanma CUDA ve hibrit için sırasıyla 2,184953 ve 1,052587 olmuştur. Ortalama hızlanmaya 128 boyut için bakıldığında CUDA 3,642001, hibrit 2,164564 değerlerini üretmiştir.

Tablo 4.8 ve Tablo 4.9, 64 ve 128 boyutta 5000 iterasyondaki sonuçları içermektedir. CUDA ve hibrit için en düşük hızlanmalar her iki boyutta f_1 fonksiyonunda gerçekleşmiştir. 64 ve 128 boyut için en düşük hızlanma sonuçları incelendiğinde sırasıyla CUDA'da 1,927629 ve 2,037173, hibritte 1,092877 ve 1,046644 olarak bulunmuştur. En yüksek hızlanmalar 64 boyutta her iki yöntem için f_3 fonksiyonunda elde edilmiştir. CUDA 9,120834, hibrit ise 3,443599 kat hızlanma değerleri sağlamıştır. 128 boyutta en yüksek hızlanma CUDA için f_4 fonksiyonunda 6,701748, hibrit için f_3 fonksiyonunda 6,493886 olarak elde edilmiştir. 64 boyutta ortalama hızlanma CUDA için 3,435801, hibrit için 1,713636 olarak hesaplanmıştır. 128 boyutta ise ortalama olarak CUDA 3,635368, hibrit 2,105599 hızlanma değeri üretmiştir.

Tablo 4.4: 64 boyut 1000 iterasyon sonuçları ve hızlanma değerleri.

Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
	CUDA	3,70E-04	3,82E-06	2,51E-02	9,38E-02	0,424827	2,200573
f_1	HİBRİT	2,63E-03	1,26E-06	3,79E-02	1,63E-02	0,795552	1,175112
	CPU	1,90E-03	3,17E-06	3,08E-02	1,93E-02	0,934863	
	CUDA	1,28E-03	7,74E-07	4,07E-02	1,27E-02	0,169593	2,522775
f_2	HİBRİT	1,34E-03	8,99E-07	1,78E-02	1,48E-02	0,288552	1,482728
	CPU	1,84E-03	2,37E-07	1,08E-02	1,30E-02	0,427844	
	CUDA	1,76E-01	1,28E-05	1,37E+00	1,62E+00	0,037711	**6,359721**
f_3	HİBRİT	3,79E-01	3,70E-05	1,70E+00	6,99E-01	0,076956	3,116508
	CPU	2,40E-01	8,79E-05	1,98E+00	7,12E-01	0,239834	
	CUDA	7,13E-02	2,99E-05	2,18E+00	8,79E-01	0,083699	3,103198
f_4	HİBRİT	4,88E-01	3,08E-05	1,72E+00	7,43E-01	0,086716	2,995214
	CPU	3,22E-01	7,94E-05	1,70E+00	6,33E-01	0,259733	
	CUDA	1,26E-04	6,79E-07	8,85E-01	1,49E-01	0,169794	2,567959
f_5	HİBRİT	9,78E-04	7,78E-06	9,88E-01	6,78E-01	0,294475	1,480683
	CPU	2,58E-04	5,79E-06	4,52E-01	1,79E-01	0,436024	
	CUDA	4,93E-04	1,13E-06	2,15E-03	5,02E-03	0,169627	3,164555
f_6	HİBRİT	8,79E-04	8,79E-06	2,75E-03	7,41E-03	0,262853	2,042183
	CPU	7,89E-05	7,89E-07	7,94E-03	4,89E-03	0,536794	
	CUDA	2,70E+00	6,99E-04	1,42E+01	4,97E+00	0,197271	3,262776
f_7	HİBRİT	2,02E+00	4,79E-04	1,03E+01	4,10E+00	0,478102	1,346263
	CPU	1,59E+00	7,89E-05	5,48E+00	4,17E+00	0,643651	
	CUDA	9,74E-03	4,24E-08	8,24E-01	7,85E-01	0,084793	5,640041
f_8	HİBRİT	8,18E-03	3,70E-08	9,63E-01	7,89E-01	0,247824	1,929744
	CPU	8,74E-03	1,89E-07	8,71E-01	1,79E-01	0,478236	
	CUDA	5,69E-03	2,46E-06	5,48E-02	9,26E-03	0,168601	2,399037
f_9	HİBRİT	7,86E-03	4,30E-06	2,79E-02	9,78E-02	0,286777	1,410434
	CPU	4,79E-03	7,89E-06	3,70E-02	4,78E-02	0,40448	
	CUDA	2,79E-03	1,62E-07	1,76E-01	6,82E-03	0,169564	2,468903
f_{10}	HİBRİT	1,89E-03	3,98E-07	1,79E-01	4,63E-01	0,29102	1,438516
	CPU	2,79E-04	5,79E-07	1,10E-01	5,79E-01	0,418637	
	CUDA	-9,90E-01	-1,00E+00	-9,04E-01	8,71E-02	0,170675	2,465065
f_{11}	HİBRİT	-9,92E-01	-1,00E+00	-9,31E-01	4,78E-02	0,283263	1,48528
	CPU	-9,94E-01	-1,00E+00	-9,43E-01	2,37E-02	0,420725	
	CUDA	2,15E-04	1,46E-11	1,21E-02	5,79E-01	0,177776	2,174073
f_{12}	HİBRİT	9,37E-04	7,89E-11	4,78E-03	6,99E-01	0,298865	1,293219
	CPU	4,32E-04	8,94E-11	7,89E-03	1,79E-01	0,386498	
	CUDA	1,78E-03	1,28E-05	1,58E-01	4,18E-01	0,176378	3,211047
f_{13}	HİBRİT	9,63E-04	8,93E-05	1,30E-01	6,02E-02	0,408437	1,386647
	CPU	8,72E-03	6,79E-05	1,00E-01	7,94E-02	0,566358	
	CUDA	1,79E-01	5,48E-04	2,28E+00	7,25E-01	0,170412	2,409965
f_{14}	HİBRİT	2,79E-01	4,78E-04	1,03E+00	7,94E-01	0,284606	1,443002
	CPU	2,69E-01	7,84E-04	1,79E+00	8,10E-01	0,410687	
	CUDA	1,15E-02	7,52E-05	1,04E-01	3,68E-02	0,204541	3,173657
f_{15}	HİBRİT	4,78E-03	7,83E-05	7,82E-02	5,40E-02	0,480324	1,351469
	CPU	1,28E-03	5,78E-05	1,78E-02	7,82E-02	0,649143	
	CUDA	3,97E-03	1,03E-05	2,11E-02	4,24E-02	0,060769	5,40186
f_{16}	HİBRİT	7,93E-04	6,32E-05	3,09E-02	5,40E-02	0,166827	1,967691
	CPU	4,79E-03	7,93E-08	2,93E-02	3,97E-02	0,328264	
	CUDA	3,40E-02	6,48E-05	9,92E-01	2,40E-01	0,175857	3,183257
f_{17}	HİBRİT	7,82E-02	4,79E-04	4,52E-01	3,99E-01	0,400918	1,396291
	CPU	1,03E-02	3,94E-04	5,98E-01	7,89E-01	0,559798	
	CUDA	2,36E-04	4,27E-07	1,29E-03	1,89E-02	0,053766	6,022602
f_{18}	HİBRİT	7,37E-04	5,78E-07	1,27E-03	3,29E-02	0,153572	2,108522
	CPU	6,27E-04	1,64E-07	2,70E-03	2,09E-02	0,32381	

Tablo 4.5: 128 boyut 1000 iterasyon sonuçları ve hızlanma değerleri.

Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
	CUDA	3,43E-03	3,67E-05	2,23E-02	3,82E-03	0,830266	2,167305
f_1	HİBRİT	1,57E-02	1,55E-04	6,57E-02	1,56E-02	1,024102	1,757091
	CPU	1,23E-02	2,38E-05	5,68E-02	1,21E-02	1,79944	
	CUDA	1,32E-02	5,75E-05	4,70E-01	4,68E-02	0,32975	2,447979
f_2	HİBRİT	4,19E-02	7,20E-05	1,87E-01	4,19E-02	0,606508	1,330932
	CPU	6,28E-03	3,11E-06	1,08E-01	1,50E-02	0,807221	
	CUDA	4,31E-01	1,23E-04	8,90E+00	1,79E+00	0,0637101	7,034139
f_3	HİBRİT	7,09E-01	4,62E-03	3,33E+00	7,01E-01	0,0628004	**7,136033**
	CPU	8,05E-01	1,90E-02	3,68E+00	7,82E-01	0,4481457	
	CUDA	7,74E-01	1,06E-02	3,43E+00	7,08E-01	0,1621502	2,727157
f_4	HİBRİT	8,28E-01	2,14E-04	3,72E+00	7,43E-01	0,163889	2,698223
	CPU	7,88E-01	1,91E-04	3,55E+00	7,89E-01	0,442209	
	CUDA	2,65E-02	6,79E-05	1,27E+00	1,27E-01	0,332183	2,499020
f_5	HİBRİT	6,21E-01	1,12E-03	2,75E+00	5,66E-01	0,6202018	1,338487
	CPU	3,81E-01	5,74E-03	1,24E+00	2,07E-01	0,830132	
	CUDA	1,73E-02	2,19E-05	2,32E-01	2,74E-02	0,332682	3,821971
f_6	HİBRİT	7,88E-03	3,25E-05	5,01E-02	8,76E-03	0,492733	2,580507
	CPU	9,79E-03	8,45E-05	4,16E-02	9,31E-03	1,271501	
	CUDA	4,85E+00	6,34E-02	3,12E+01	4,10E+00	0,384055	3,425582
f_7	HİBRİT	4,70E+00	5,67E-02	2,66E+01	4,32E+00	0,7228125	1,820129
	CPU	4,53E+00	6,54E-02	1,70E+01	4,13E+00	1,3156119	
	CUDA	5,16E-01	7,73E-04	2,72E+00	1,73E-01	0,164907	4,331951
f_8	HİBRİT	6,23E-01	5,56E-03	2,89E+00	6,00E-01	0,530821	1,345781
	CPU	2,67E-02	1,28E-06	1,11E+00	1,28E-01	0,714369	
	CUDA	4,19E-02	1,31E-04	5,42E-01	1,37E-02	0,32453	2,421667
f_9	HİBRİT	9,04E-02	3,91E-05	3,67E-01	8,88E-02	0,7420253	1,059133
	CPU	6,83E-02	4,51E-04	2,16E-01	5,52E-01	0,7859037	
	CUDA	5,41E-03	1,87E-05	5,58E-01	7,50E-03	0,329177	2,384460
f_{10}	HİBRİT	8,69E-02	1,07E-03	4,42E-01	8,08E-02	0,6118542	1,282837
	CPU	7,61E-02	3,41E-04	4,94E-01	8,06E-02	0,7849095	
	CUDA	-9,58E-01	-1,00E+00	-8,76E-01	7,00E-02	0,331358	2,303469
f_{11}	HİBRİT	-9,75E-01	-1,00E+00	-8,77E-01	2,36E-02	0,567893	1,344044
	CPU	-9,75E-01	-1,00E+00	-8,93E-01	2,37E-02	0,763273	
	CUDA	4,96E-03	3,78E-05	1,84E-02	4,19E-03	0,345736	2,254596
f_{12}	HİBRİT	7,80E-03	6,32E-05	3,69E-02	7,23E-03	0,63544658	1,226689
	CPU	4,32E-04	1,97E-10	1,08E-02	1,70E-03	0,7794951	
	CUDA	2,48E-01	2,75E-04	3,73E+00	5,14E-01	0,345796	3,101460
f_{13}	HİBRİT	7,22E-02	2,76E-04	3,92E-01	6,97E-02	0,6404238	1,674629
	CPU	6,39E-02	5,72E-04	3,07E-01	6,88E-02	1,0724723	
	CUDA	3,09E-01	1,24E-03	4,62E+00	7,39E-01	0,329281	2,269576
f_{14}	HİBRİT	8,48E-01	9,85E-03	5,00E+00	8,75E-01	0,593298	1,259617
	CPU	7,37E-01	2,58E-03	4,63E+00	7,45E-01	0,7473281	
	CUDA	3,50E-02	1,74E-04	2,27E-01	3,84E-02	0,401729	3,142629
f_{15}	HİBRİT	4,60E-02	1,83E-03	1,92E-01	3,98E-02	0,7898882	1,598309
	CPU	5,44E-02	2,73E-04	4,10E-01	5,74E-02	1,2624851	
	CUDA	7,29E-03	6,41E-05	5,00E-02	8,24E-03	0,102009	6,345424
f_{16}	HİBRİT	8,68E-02	2,32E-03	3,48E-01	7,51E-02	0,1653028	3,915786
	CPU	4,70E-02	6,85E-07	3,08E-01	6,08E-02	0,6472904	
	CUDA	2,77E-01	3,26E-03	2,34E+00	2,92E-01	0,343738	3,196256
f_{17}	HİBRİT	2,55E-01	8,51E-03	1,17E+00	2,46E-01	0,6270792	1,752051
	CPU	2,10E-01	4,88E-03	1,38E+00	2,33E-01	1,0986746	
	CUDA	3,29E-02	1,14E-05	6,83E-01	7,73E-03	0,0969718	6,413972
f_{18}	HİBRİT	4,04E-02	3,97E-04	2,64E-01	4,19E-02	0,1215527	5,116911
	CPU	1,28E-02	1,64E-06	1,18E-01	2,13E-02	0,6219744	

Tablo 4.6: 64 boyut 2500 iterasyon sonuçları ve hızlanma değerleri.

Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
f_1	CUDA	2,63E-03	6,53E-06	2,80E-02	5,82E-02	1,0604	1,942267
	HİBRİT	3,40E-04	1,32E-06	2,37E-02	2,02E-02	1,8933	1,087825
	CPU	1,79E-03	3,79E-06	2,79E-02	3,98E-02	2,05958	
f_2	CUDA	1,48E-03	6,25E-07	3,92E-02	1,40E-02	0,424364	2,171878
	HİBRİT	1,02E-03	4,03E-07	1,40E-02	1,79E-02	0,673175	1,369134
	CPU	1,40E-03	5,79E-07	1,01E-02	1,54E-02	0,921667	
f_3	CUDA	1,78E-01	2,74E-05	1,21E+00	1,79E+00	0,081087	**7,619271**
	HİBRİT	1,28E-01	0,00E+00	1,40E+00	8,04E-01	0,17827	3,465646
	CPU	2,01E-01	1,01E-05	1,78E+00	6,79E-01	0,61782	
f_4	CUDA	6,98E-02	3,03E-05	2,02E+00	8,10E-01	0,170542	3,070681
	HİBRİT	5,79E-02	4,79E-05	1,69E+00	7,78E-01	0,172478	3,036208
	CPU	2,48E-01	2,88E-05	1,58E+00	6,25E-01	0,52368	
f_5	CUDA	1,10E-04	6,40E-07	8,79E-01	1,58E-01	0,428349	2,438969
	HİBRİT	4,78E-04	5,88E-06	7,93E-01	5,98E-01	0,698902	1,494816
	CPU	1,03E-04	4,99E-06	4,40E-01	1,48E-01	1,04473	
f_6	CUDA	4,77E-04	5,96E-07	1,55E-03	5,59E-03	0,426753	3,06306
	HİBRİT	7,52E-04	5,79E-06	2,28E-03	4,93E-03	0,591425	2,210204
	CPU	3,28E-05	5,78E-07	6,74E-03	3,97E-03	1,30717	
f_7	CUDA	2,43E+00	6,79E-04	1,46E+01	5,07E+00	0,4943	3,101962
	HİBRİT	1,99E+00	5,79E-04	9,25E+00	4,20E+00	1,14277	1,34174
	CPU	1,79E+00	6,52E-06	4,98E+00	4,79E+00	1,5333	
f_8	CUDA	9,99E-03	1,88E-07	7,42E-01	5,52E-01	0,20278	5,05393
	HİBRİT	6,22E-03	4,79E-08	8,24E-01	5,79E-01	0,698753	1,466664
	CPU	1,00E-03	1,48E-07	3,00E-01	2,02E-01	1,024836	
f_9	CUDA	5,78E-03	2,15E-06	5,03E-02	9,18E-03	0,421325	2,387373
	HİBRİT	5,79E-03	4,01E-06	1,99E-02	8,75E-02	0,682402	1,473999
	CPU	4,18E-03	5,79E-06	3,48E-02	5,02E-02	1,00586	
f_{10}	CUDA	3,79E-03	1,74E-07	2,05E-01	6,32E-01	0,424146	2,436873
	HİBRİT	2,79E-03	1,18E-06	1,37E-01	5,79E-01	0,690124	1,497687
	CPU	1,30E-04	6,98E-07	1,02E-01	4,98E-01	1,03359	
f_{11}	CUDA	-9,98E-01	-1,00E+00	-9,82E-01	5,79E-02	0,426259	2,353991
	HİBRİT	-9,90E-01	-1,00E+00	-9,87E-01	3,78E-02	0,672807	1,491379
	CPU	-9,97E-01	-1,00E+00	-9,74E-01	4,78E-02	1,00341	
f_{12}	CUDA	2,02E-04	0,00E+00	1,40E-03	5,05E-01	0,444794	1,964071
	HİBRİT	5,21E-04	1,04E-10	6,72E-03	3,98E-01	0,709053	1,232076
	CPU	5,78E-04	4,89E-11	4,98E-03	1,08E-01	0,873607	
f_{13}	CUDA	1,54E-03	1,32E-05	1,97E-01	4,02E-01	0,441599	2,989182
	HİBRİT	7,94E-03	6,32E-05	1,00E-01	3,98E-02	0,943806	1,398614
	CPU	5,21E-03	1,79E-05	1,02E-01	4,78E-02	1,32002	
f_{14}	CUDA	1,87E-01	4,78E-04	2,29E+00	7,68E-01	0,424402	2,287496
	HİBRİT	2,02E-01	1,03E-04	1,00E+00	6,78E-01	0,673498	1,441456
	CPU	3,88E-01	6,98E-04	1,01E+00	7,00E-01	0,970818	
f_{15}	CUDA	1,25E-02	7,65E-05	1,08E-01	3,53E-02	0,51366	3,011759
	HİBRİT	3,10E-03	4,78E-05	3,99E-02	6,98E-02	1,15997	1,333672
	CPU	1,00E-03	4,99E-05	1,40E-02	7,24E-02	1,54702	
f_{16}	CUDA	3,87E-03	1,25E-07	2,33E-02	4,14E-02	0,132479	5,839008
	HİBRİT	6,99E-04	1,08E-07	2,79E-02	6,99E-02	0,36838	2,099859
	CPU	3,30E-03	5,79E-08	2,70E-02	2,18E-02	0,773546	
f_{17}	CUDA	3,40E-02	6,48E-05	9,92E-01	2,40E-01	0,175857	3,183257
	HİBRİT	7,82E-02	4,79E-04	4,52E-01	3,99E-01	0,400918	1,396291
	CPU	1,03E-02	3,94E-04	5,98E-01	7,89E-01	0,559798	
f_{18}	CUDA	8,48E-04	3,28E-07	1,01E-03	1,78E-02	0,118437	5,977752
	HİBRİT	5,08E-04	2,37E-08	5,03E-03	3,29E-02	0,303981	2,32905
	CPU	1,79E-04	3,59E-08	6,52E-03	1,89E-02	0,707987	

Tablo 4.7: 128 boyut 2500 iterasyon sonuçları ve hızlanma değerleri.

Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
	CUDA	2,35E-01	2,37E-02	7,12E-05	2,89E-02	2,07252	2,184953
f_1	HİBRİT	9,09E-02	1,64E-01	5,17E-05	1,61E-02	4,302124	1,052587
	CPU	6,22E-02	9,22E-03	9,58E-05	1,03E-02	4,528359	
	CUDA	4,27E-01	5,50E-02	5,02E-04	8,40E-02	0,823729	2,431898
f_2	HİBRİT	1,51E-01	3,95E-02	1,65E-04	3,83E-02	1,51919	1,318614
	CPU	1,18E-01	3,95E-02	6,04E-05	1,74E-02	2,003225	
	CUDA	7,44E+00	2,05E+00	1,24E-02	1,79E+00	0,150475	6,976619
f_3	HİBRİT	3,79E+00	7,76E-01	1,12E-02	6,88E-01	0,149716	**7,012002**
	CPU	2,99E+00	6,70E-01	0,00E+00	6,95E-01	1,049809	
	CUDA	8,96E+00	1,58E+00	4,14E-04	2,49E+00	0,148197	6,708388
f_4	HİBRİT	3,55E+00	7,37E-01	6,13E-03	6,20E-01	0,40841	2,434228
	CPU	3,19E+00	7,74E-01	1,08E-02	7,23E-01	0,994163	
	CUDA	1,25E+01	1,14E+00	8,91E-03	1,65E+00	0,830028	2,440997
f_5	HİBRİT	4,39E+00	8,66E-01	9,13E-03	7,66E-01	1,514622	1,337691
	CPU	1,52E+00	3,50E-01	1,68E-03	2,42E-01	2,026096	
	CUDA	1,48E-01	1,79E-02	1,30E-03	1,79E-02	0,813648	2,779039
f_6	HİBRİT	3,56E-02	1,09E-02	1,71E-04	8,48E-03	1,21335	1,863568
	CPU	4,45E-02	9,33E-03	1,59E-04	8,87E-03	2,26116	
	CUDA	1,85E+01	4,71E+00	6,03E-02	4,49E+00	0,959649	3,420429
f_7	HİBRİT	3,15E+01	4,77E+00	5,91E-02	5,10E+00	1,814355	1,809134
	CPU	2,46E+01	4,75E+00	1,77E-01	4,70E+00	3,282411	
	CUDA	3,13E+00	7,13E-01	1,10E-03	6,57E-01	0,35483	5,118705
f_8	HİBRİT	4,59E+00	8,67E-01	9,61E-03	9,35E-01	1,21839	1,490713
	CPU	2,39E+00	7,00E-02	2,04E-07	3,16E-01	1,81627	
	CUDA	5,01E-01	1,41E-01	1,28E-03	1,18E-01	0,804263	2,328868
f_9	HİBRİT	4,05E-01	8,30E-02	1,15E-03	7,89E-02	1,489586	1,257411
	CPU	3,25E-01	7,75E-02	1,04E-04	6,80E-02	1,873022	
	CUDA	7,14E-01	1,64E-01	7,68E-05	1,40E-01	0,810489	2,473513
f_{10}	HİBRİT	3,33E-01	8,50E-02	3,85E-04	7,70E-02	1,556416	1,288059
	CPU	4,47E-01	7,56E-02	2,52E-04	7,56E-02	2,004755	
	CUDA	-8,50E-01	-9,79E-01	-1,00E+00	5,79E-02	0,825288	2,427859
f_{11}	HİBRİT	-8,61E-01	-9,73E-01	-1,00E+00	2,61E-02	1,459358	1,372989
	CPU	-8,44E-01	-9,77E-01	-1,00E+00	2,50E-02	2,003683	
	CUDA	9,79E-02	7,10E-03	7,94E-05	1,24E-02	0,861783	2,246086
f_{12}	HİBRİT	4,33E-02	8,33E-03	1,48E-04	8,35E-03	1,586536	1,220041
	CPU	1,89E-02	4,50E-04	3,07E-10	2,30E-03	1,935639	
	CUDA	1,14E+00	1,79E-01	2,68E-04	1,48E-01	0,850178	3,291354
f_{13}	HİBRİT	2,70E-01	7,56E-02	7,92E-04	6,63E-02	1,616467	1,731082
	CPU	3,13E-01	6,61E-02	1,48E-03	6,51E-02	2,798237	
	CUDA	4,91E+00	7,04E-01	1,40E-03	7,60E-01	0,81412	2,224405
f_{14}	HİBRİT	4,84E+00	7,69E-01	1,63E-03	8,01E-01	1,466275	1,235057
	CPU	2,47E+00	5,78E-01	7,40E-03	5,43E-01	1,810933	
	CUDA	2,15E-01	4,40E-02	2,14E-05	4,37E-02	0,996397	2,984781
f_{15}	HİBRİT	2,67E-01	5,21E-02	1,87E-03	5,16E-02	1,854263	1,603886
	CPU	2,11E-01	4,63E-02	1,70E-04	4,62E-02	2,974027	
	CUDA	2,16E-01	5,92E-02	1,29E-03	5,78E-02	0,253174	5,815518
f_{16}	HİBRİT	5,30E-01	7,82E-02	1,19E-03	8,07E-02	0,35657	4,129169
	CPU	2,64E-01	4,90E-02	3,03E-06	5,29E-02	1,472336	
	CUDA	2,06E+00	2,40E-01	1,02E-03	2,79E-01	0,859182	3,227087
f_{17}	HİBRİT	1,29E+00	2,94E-01	3,47E-03	2,87E-01	1,602434	1,730277
	CPU	1,31E+00	2,50E-01	1,29E-03	2,70E-01	2,772655	
	CUDA	1,72E-01	3,79E-02	1,13E-03	3,52E-02	0,231763	6,475525
f_{18}	HİBRİT	2,09E-01	4,05E-02	1,44E-04	3,94E-02	0,2957	5,075366
	CPU	1,53E-01	1,05E-02	1,20E-06	2,16E-02	1,500787	

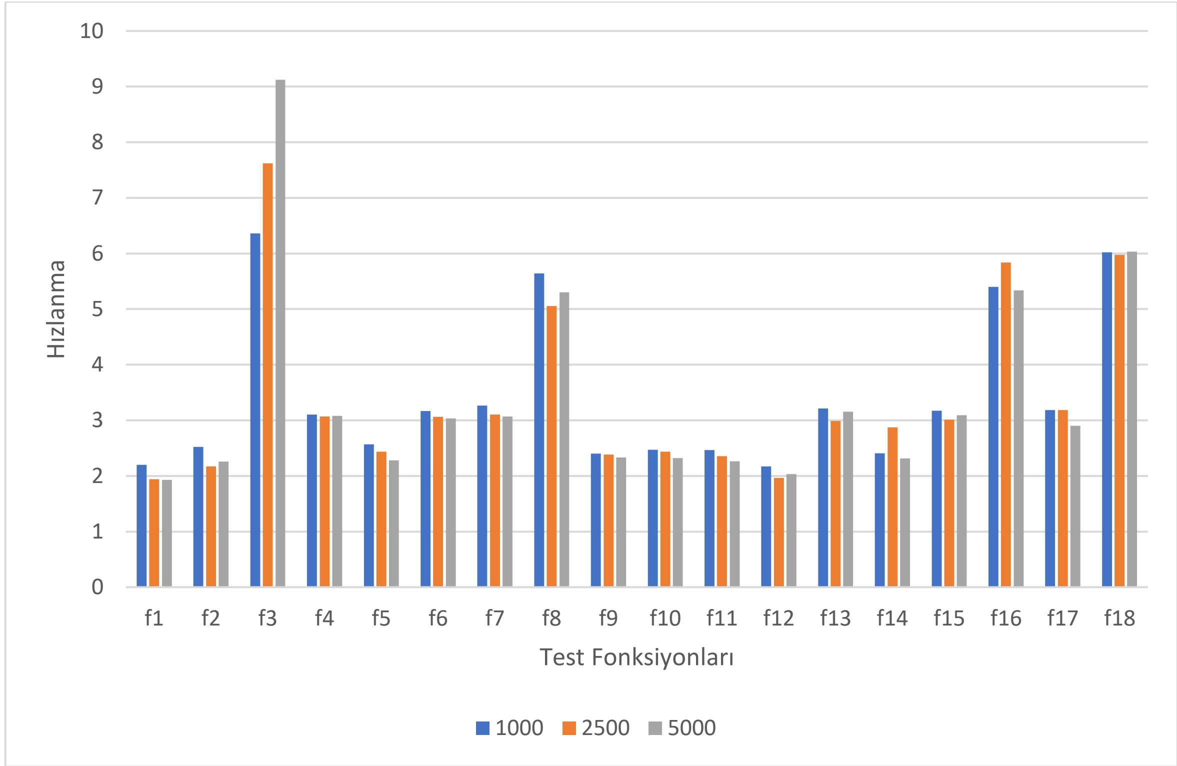
Tablo 4.8: 64 boyut 5000 iterasyon sonuçları ve hızlanma değerleri.

Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
f_1	CUDA	2,85E-03	5,33E-06	1,97E-02	1,73E-02	2,11949	1,927629
	HİBRİT	1,48E-03	1,04E-06	1,99E-02	2,18E-02	3,73838	1,092877
	CPU	1,03E-04	1,84E-06	2,99E-02	5,93E-02	4,08559	
f_2	CUDA	1,78E-03	5,79E-07	8,94E-03	1,40E-02	0,849132	2,256752
	HİBRİT	1,03E-03	1,97E-06	1,03E-02	2,79E-02	1,33625	1,434073
	CPU	1,08E-03	8,02E-07	1,28E-02	6,18E-02	1,91628	
f_3	CUDA	1,62E-01	0,00E+00	1,88E+00	1,40E+00	0,162104	**9,120834**
	HİBRİT	1,40E-01	6,01E-06	1,20E+00	9,02E-01	0,429354	3,443599
	CPU	2,79E-01	2,79E-05	1,62E+00	8,93E-01	1,478524	
f_4	CUDA	9,37E-02	3,79E-05	2,28E+00	8,79E-01	0,421783	3,082562
	HİBRİT	3,79E-01	6,00E-05	1,99E+00	9,00E-01	0,478236	2,718683
	CPU	2,00E-01	1,98E-05	1,00E+00	5,79E-01	1,300172	
f_5	CUDA	1,28E-04	8,75E-07	8,60E-01	1,79E-01	0,8549	2,278138
	HİBRİT	3,98E-04	4,98E-06	6,97E-01	7,89E-01	1,36119	1,430792
	CPU	4,78E-04	5,70E-06	4,40E-01	3,02E-01	1,94758	
f_6	CUDA	4,24E-04	5,64E-07	2,36E-03	4,08E-03	0,853974	3,034589
	HİBRİT	6,30E-04	7,29E-06	4,00E-03	9,18E-03	1,21846	2,126832
	CPU	6,01E-05	9,00E-07	5,74E-03	6,19E-03	2,59146	
f_7	CUDA	2,63E+00	7,00E-04	1,58E+01	4,98E+00	1,0158	3,067385
	HİBRİT	2,00E+00	4,24E-04	7,98E+00	4,79E+00	2,29775	1,356044
	CPU	1,62E+00	1,03E-06	3,79E+00	5,00E+00	3,11585	
f_8	CUDA	8,02E-03	1,98E-07	7,36E-01	8,79E-01	0,423987	5,301694
	HİBRİT	3,08E-03	1,29E-07	9,21E-01	7,00E-01	1,578362	1,424166
	CPU	1,02E-03	1,68E-07	4,79E-01	5,18E-01	2,24785	
f_9	CUDA	8,96E-03	1,37E-06	6,22E-02	8,73E-03	0,878563	2,330055
	HİBRİT	6,00E-03	2,18E-06	1,01E-02	9,99E-02	1,33452	1,53396
	CPU	3,79E-03	6,98E-06	5,79E-02	6,01E-02	2,0471	
f_{10}	CUDA	2,69E-03	1,94E-07	2,79E-01	7,83E-01	0,848357	2,319943
	HİBRİT	1,79E-03	2,37E-07	1,02E-01	7,93E-01	1,35562	1,451838
	CPU	1,10E-04	8,02E-07	1,00E-01	5,99E-01	1,96814	
f_{11}	CUDA	-9,96E-01	-1,00E+00	-9,43E-01	8,93E-03	0,850932	2,260745
	HİBRİT	-9,97E-01	-1,00E+00	-9,72E-01	5,02E-02	1,31463	1,463332
	CPU	-9,91E-01	-1,00E+00	-9,33E-01	6,78E-02	1,92374	
f_{12}	CUDA	9,66E-04	0,00E+00	4,75E-03	1,09E-01	0,895626	2,03039
	HİBRİT	7,24E-04	3,97E-11	9,00E-03	7,00E-01	1,41105	1,288735
	CPU	6,00E-04	7,33E-11	5,79E-03	3,63E-01	1,81847	
f_{13}	CUDA	1,28E-03	1,48E-04	1,84E-01	4,78E-01	0,868193	3,155992
	HİBRİT	7,00E-03	1,90E-05	1,37E-01	8,24E-02	1,93182	1,418357
	CPU	8,00E-03	1,00E-05	1,42E-01	1,00E-01	2,74001	
f_{14}	CUDA	1,24E-01	4,46E-04	2,00E+00	8,00E-01	0,833107	2,317457
	HİBRİT	1,99E-01	2,00E-04	1,18E+00	7,82E-01	1,32631	1,455685
	CPU	1,00E-01	8,00E-04	1,20E+00	7,63E-01	1,93069	
f_{15}	CUDA	1,02E-02	8,00E-05	1,26E-01	3,70E-02	1,01017	3,093192
	HİBRİT	2,78E-03	6,00E-05	6,97E-02	5,60E-02	2,2956	1,361147
	CPU	1,28E-03	3,40E-05	1,00E-02	7,00E-02	3,12465	
f_{16}	CUDA	4,02E-03	1,00E-08	3,70E-02	4,98E-02	0,285551	5,336105
	HİBRİT	8,02E-04	2,10E-08	1,20E-02	9,02E-02	0,724524	2,103077
	CPU	4,00E-03	9,00E-08	3,02E-02	7,82E-02	1,52373	
f_{17}	CUDA	2,78E-02	7,01E-05	9,91E-01	2,65E-01	0,885586	2,899797
	HİBRİT	7,99E-02	2,49E-04	2,98E-01	4,00E-01	1,74314	1,473215
	CPU	1,48E-03	8,75E-05	2,01E-01	9,52E-01	2,56802	
f_{18}	CUDA	1,40E-04	3,84E-07	8,27E-03	1,98E-02	0,236785	6,031168
	HİBRİT	8,01E-04	1,79E-07	5,32E-03	2,97E-02	0,629383	2,269032
	CPU	4,00E-04	2,79E-07	7,85E-03	1,98E-02	1,42809	

Tablo 4.9: 128 boyut 5000 iterasyon sonuçları ve hızlanma değerleri.

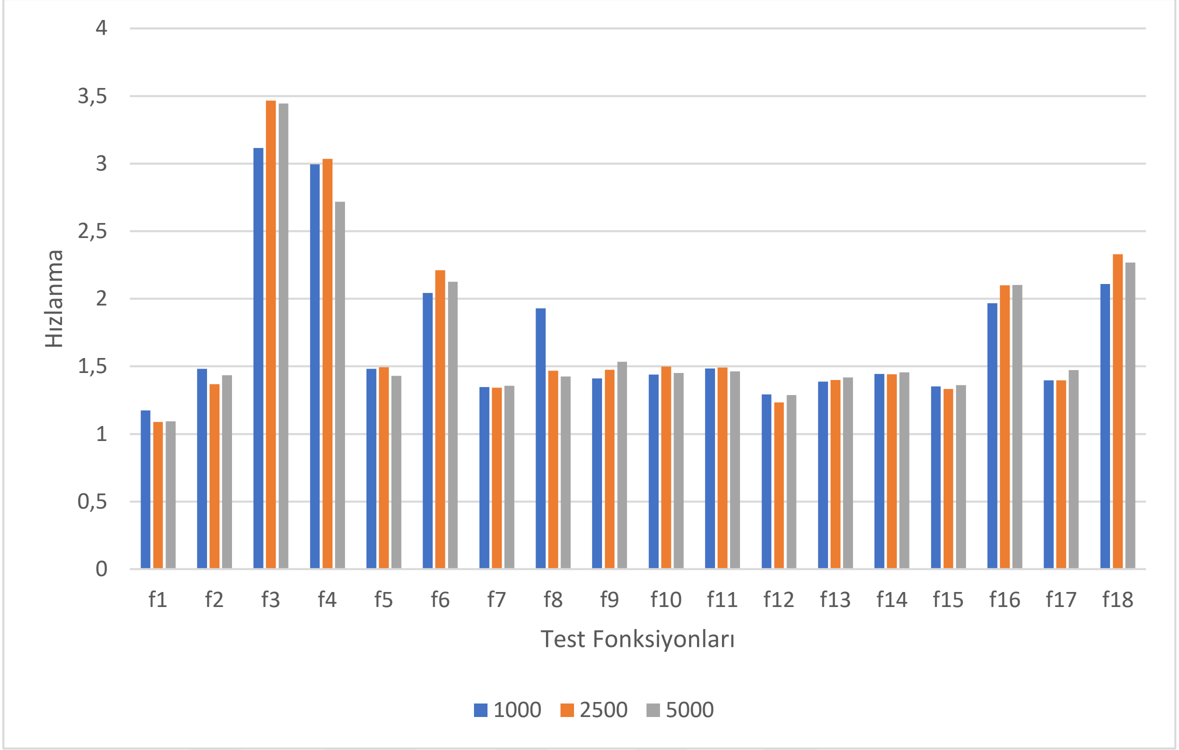
Fonksiyon Numarası	Yöntem	Ortalama	En İyi	En Kötü	Standart Sapma	Ortalama Hesaplama Süresi (Saniye)	Hızlanma
	CUDA	3,37E-02	1,23E-05	2,79E-01	4,68E-02	4,1347	2,037173
f_1	HİBRİT	1,75E-02	3,89E-05	1,26E-01	1,82E-02	8,047724	1,046644
	CPU	8,78E-03	1,57E-05	5,75E-02	1,03E-02	8,423101	
	CUDA	4,58E-02	7,37E-05	3,72E-01	5,65E-02	1,64398	2,245704
f_2	HİBRİT	4,52E-02	5,10E-04	2,40E-01	4,30E-02	2,990212	1,234659
	CPU	5,91E-03	6,63E-05	8,27E-02	1,25E-02	3,691893	
	CUDA	1,24E+00	8,14E-02	8,27E+00	1,40E+00	0,31157	6,184992
f_3	HİBRİT	6,07E-01	1,32E-03	2,77E+00	5,72E-01	0,29675	6,493886
	CPU	7,28E-01	9,22E-03	3,98E+00	6,84E-01	1,927058	
	CUDA	6,98E-01	1,28E-03	4,48E+00	8,65E-01	0,296357	**6,701748**
f_4	HİBRİT	7,04E-01	8,79E-03	3,20E+00	7,31E-01	0,79446	2,49995
	CPU	6,99E-01	2,34E-02	4,39E+00	7,50E-01	1,98611	
	CUDA	1,40E+00	1,11E-02	9,00E+00	1,51E+00	1,6481	2,291549
f_5	HİBRİT	7,87E-01	7,93E-04	3,53E+00	7,46E-01	3,019911	1,2506
	CPU	3,91E-01	1,95E-02	1,11E+00	2,17E-01	3,776702	
	CUDA	1,87E-02	3,33E-04	1,10E-01	1,85E-02	1,63914	2,857846
f_6	HİBRİT	9,59E-03	1,92E-05	7,39E-02	1,03E-02	2,38817	1,961506
	CPU	9,29E-03	4,35E-05	4,73E-02	9,65E-03	4,68441	
	CUDA	5,02E+00	1,98E-01	3,02E+01	5,02E+00	1,96823	3,259793
f_7	HİBRİT	4,91E+00	7,04E-02	1,83E+01	4,36E+00	3,635718	1,76472
	CPU	4,36E+00	1,92E-02	1,36E+01	3,64E+00	6,416022	
	CUDA	6,53E-01	2,63E-03	6,00E+00	8,94E-01	0,69823	5,158673
f_8	HİBRİT	7,07E-01	1,73E-02	5,23E+00	8,10E-01	2,3251	1,549155
	CPU	8,17E-02	8,64E-07	1,64E+00	2,53E-01	3,60194	
	CUDA	1,05E-01	1,02E-03	6,18E-01	1,13E-01	1,63367	2,364737
f_9	HİBRİT	9,14E-02	2,47E-04	4,20E-01	8,43E-02	2,993636	1,290471
	CPU	1,02E-01	4,73E-04	4,10E-01	7,38E-02	3,8632	
	CUDA	1,48E-01	8,94E-04	5,94E-01	1,25E-01	1,64478	2,399607
f_{10}	HİBRİT	7,12E-02	7,42E-04	3,78E-01	6,87E-02	3,024314	1,305031
	CPU	7,72E-02	1,91E-06	4,45E-01	7,49E-02	3,946825	
	CUDA	-9,80E-01	-1,00E+00	-8,93E-01	7,35E-03	1,6471	2,280517
f_{11}	HİBRİT	-9,85E-01	-1,00E+00	-8,84E-01	1,59E-02	3,33588	1,126012
	CPU	-9,77E-01	-1,00E+00	-8,92E-01	2,40E-02	3,75624	
	CUDA	7,10E-03	7,94E-05	9,79E-02	1,24E-02	0,861783	2,246086
f_{12}	HİBRİT	8,33E-03	1,48E-04	4,33E-02	8,35E-03	1,586536	1,220041
	CPU	4,50E-04	3,07E-10	1,89E-02	2,30E-03	1,935639	
	CUDA	9,03E-02	7,86E-03	5,13E-01	9,40E-02	1,72719	3,210805
f_{13}	HİBRİT	7,35E-02	1,02E-03	3,94E-01	7,71E-02	3,398929	1,631594
	CPU	8,29E-02	3,69E-03	4,53E-01	8,66E-02	5,545671	
	CUDA	7,24E-01	3,79E-03	9,04E+00	9,89E-01	1,64044	2,396892
f_{14}	HİBRİT	8,07E-01	9,35E-03	3,58E+00	7,29E-01	2,963777	1,326671
	CPU	6,70E-01	1,22E-02	4,83E+00	7,79E-01	3,931958	
	CUDA	3,70E-02	1,70E-03	1,40E-01	3,42E-01	1,99433	3,293727
f_{15}	HİBRİT	5,58E-02	1,45E-04	3,84E-01	6,27E-02	4,915683	1,33629
	CPU	6,72E-02	7,67E-04	3,00E-01	6,63E-02	6,568778	
	CUDA	3,70E-02	6,98E-04	2,05E-01	4,03E-02	0,505369	5,874541
f_{16}	HİBRİT	9,08E-02	1,30E-03	4,34E-01	8,01E-02	0,715044	4,151929
	CPU	4,19E-02	1,81E-07	2,36E-01	5,47E-02	2,968811	
	CUDA	3,46E-01	1,70E-03	2,00E+00	4,02E-01	1,171796	4,306946
f_{17}	HİBRİT	2,32E-01	2,30E-03	1,39E+00	2,24E-01	3,140508	1,607021
	CPU	2,48E-01	8,53E-04	1,33E+00	2,42E-01	5,046862	
	CUDA	3,87E-02	2,79E-04	2,97E-01	4,79E-02	0,459236	6,325287
f_{18}	HİBRİT	3,97E-01	1,45E-03	2,62E-01	3,62E-02	0,569055	5,1046
	CPU	1,12E-02	1,30E-06	9,51E-02	2,03E-02	2,9048	

Çalışmanın bu aşamasında, her iki yöntemde 64 ve 128 boyut için her bir test fonksiyonunun hızlanma değerleri grafiksel olarak verilmektedir. Şekil 4.7'de 64 boyutlu problemin CUDA paralel uygulamasının 1000, 2500 ve 5000 iterasyon için hızlanma sonuçları sunulmuştur. Şekilde görüldüğü üzere en yüksek hızlanma f_3 fonksiyonunda 5000 iterasyon için 9,120834 olarak elde edilmiştir. En düşük hızlanma ise f_1 fonksiyonunda 5000 iterasyon için 1,927629 olarak bulunmuştur.



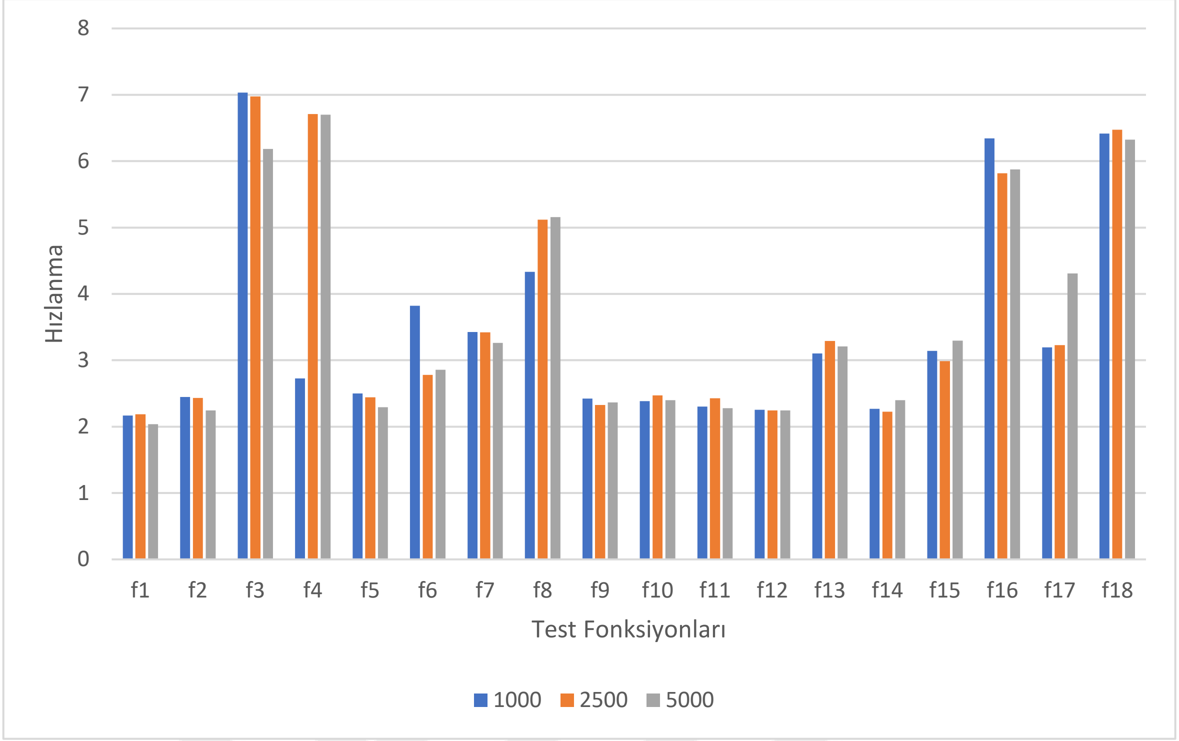
Şekil 4.7: 64 boyut için CUDA hızlanma grafiği.

Şekil 4.8'de 64 boyutlu problemin çözümünde kullanılan hibrit yöntemin iterasyon bazında hızlanma sonuçları verilmiştir. Sonuçlar incelendiğinde hem en yüksek hem de en düşük hızlanma 2500 iterasyon için elde edilmiştir. Şekilde görüldüğü gibi bu yöntemde en yüksek hızlanma f_3 fonksiyonunda 3,465646, en düşük hızlanma ise f_1 fonksiyonunda 1,087825 olarak hesaplanmıştır.



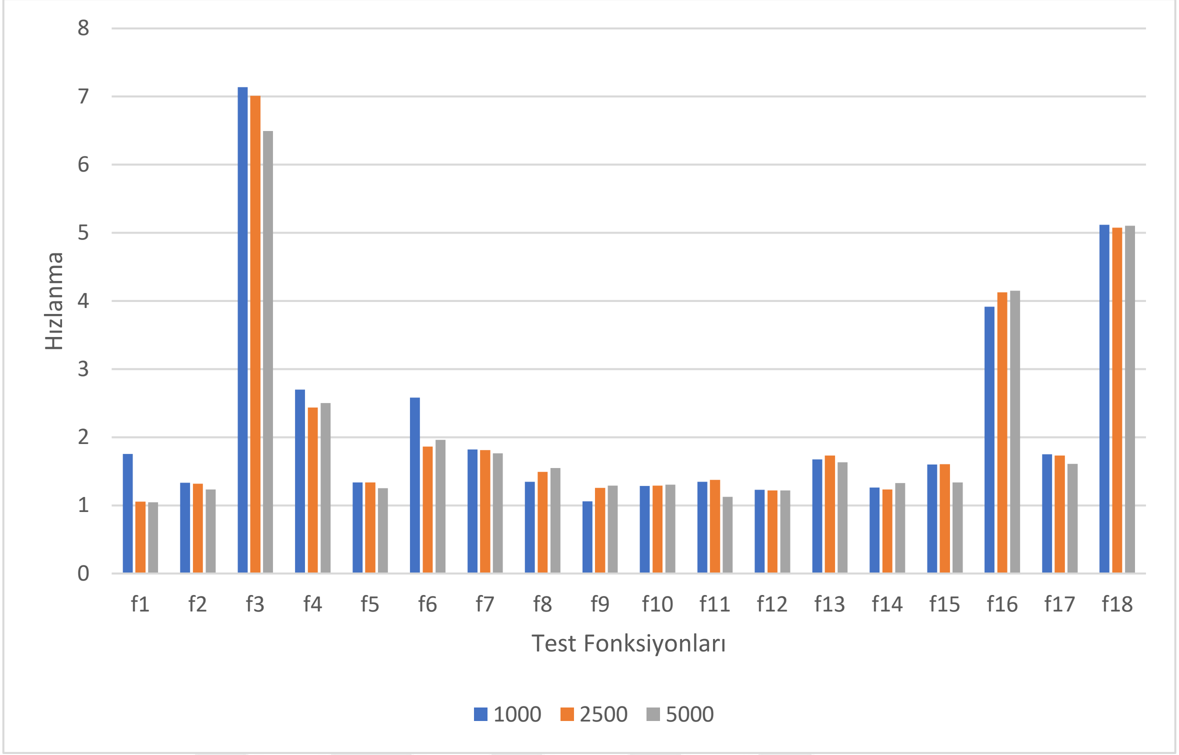
Şekil 4.8: 64 boyut için hibrit hızlanma grafiği.

Şekil 4.9'da 128 boyut için CUDA uygulamasının hızlanma sonuçları sunulmuştur. Sonuçlar analiz edildiğinde en yüksek hızlanma f_3 fonksiyonunda 1000 iterasyonda 7,034139 olarak elde edilmiştir. En düşük hızlanma f_1 fonksiyonunda 5000 iterasyonda 2,037173 hesaplanmıştır.



Şekil 4.9: 128 boyut için CUDA hızlanma grafiği.

Şekil 4.10'da 128 boyut için hibrit yöntemle elde edilen hızlanma değerleri verilmektedir. Elde edilen sonuçlara göre en yüksek hızlanma f_3 fonksiyonunda 1000 iterasyonda 7,136033 en düşük hızlanma ise f_1 fonksiyonunda 5000 iterasyonda 1,046644 olarak bulunmuştur.



Şekil 4.10: 128 boyut için hibrit hızlanma grafiği.

Tüm şekiller birlikte incelendiğinde her iki boyut için her iki yöntemde en düşük (f_1 fonksiyonunda) ve en yüksek (f_3 fonksiyonunda) hızlanmaların aynı fonksiyonlar üzerinde gerçekleştiği görülmektedir. Bu sonuç fonksiyonların yapısının elde edilen hızlanma değerlerinde etkili olduğunu göstermektedir. İterasyon sayısındaki değişimlere bakıldığında, hızlanmaya etkisinin genellikle düşük olduğu görülmüştür.

Tablo 4.10'da 64 boyutlu problemlerde farklı iterasyon sayıları için elde edilen en yüksek, en düşük ve ortalama hızlanma sonuçları verilmektedir. CUDA çözümlerde 1000 iterasyon için 2,174073 ile 6,359721 arasında, 2500 iterasyon için 1,942267 ile 7,619271 arasında ve 5000 iterasyon için 1,927629 ile 9,120834 arasında hızlanma elde edilmiştir. Tüm test fonksiyonlarından CUDA kullanılarak elde edilen hızlanma sonuçlarının ortalamasına bakıldığında iterasyon sayısına göre sırasıyla 3,429504, 3,415354 ve 3,435801 değerleri görülmektedir. Hibrit yöntemde, 1000 iterasyon için 1,175112 ile 3,116508 arasında, 2500 iterasyon için 1,087825 ile 3,465646 arasında ve 5000 iterasyon için 1,092877 ile 3,443599 arasında hızlanma elde edilmiştir. Hibrit ortalama sonuçları ise iterasyon sırasıyla 1,713861, 1,731461 ve 1,713636 olarak hesaplanmıştır. Ortalama sonuçları birlikte incelendiğinde, CUDA'nın hibrit yönteme göre yaklaşık 2 kat daha yüksek hızlanma ürettiği görülmektedir.

Tablo 4.10: 64 boyut için CUDA ve hibrit hızlanma değerleri.

Hızlanma	Yöntem	İterasyon		
		1000	2500	5000
En yüksek	CUDA	6,359721	7,619271	9,120834
	Hibrit	3,116508	3,465646	3,443599
En düşük	CUDA	2,174073	1,942267	1,927629
	Hibrit	1,175112	1,087825	1,092877
Ortalama	CUDA	3,429504	3,415354	3,435801
	Hibrit	1,713861	1,731461	1,713636

Tablo 4.11'de 128 boyutlu problemlerin CUDA ve hibrit hızlanma sonuçları verilmiştir. Sonuçlar incelendiğinde CUDA yönteminde hızlanma değerleri 1000 iterasyonda 2,167305 ile 7,034139, 2500 iterasyonda 1,05287 ile 6,976619 ve 5000 iterasyonda 2,037173 ile 6,701748 arasında hesaplanmıştır. Hibrit yöntemde ise 1000 iterasyonda 1,059133 ile 7,136033, 2500 iterasyonda 1,05287 ile 7,012002 ve 5000 iterasyonda 1,046644 ile 6,493886 aralığında olmak üzere hızlanmalar elde edilmiştir. Ortalama hızlanma değerleri iterasyon sırasıyla CUDA yönteminde 3,460499, 3,642001 ve 3,635368 olarak hibrit yöntemde ise 2,235399, 2,164564 ve 2,105599 olarak görülmektedir. Ortalama sonuçlara her iki yöntemde hızlanma değerlerine birlikte bakıldığında hibrit yöntemin CUDA'ya göre yaklaşık 1,5 kat yavaş olduğu görülmektedir.

Tablo 4.11: 128 boyut için CUDA ve hibrit hızlanma değerleri.

Hızlanma	Yöntem	İterasyon		
		1000	2500	5000
En yüksek	CUDA	7,034139	6,976619	6,701748
	Hibrit	7,136033	7,012002	6,493886
En düşük	CUDA	2,167305	2,184953	2,037173
	Hibrit	1,059133	1,05287	1,046644
Ortalama	CUDA	3,460499	3,642001	3,635368
	Hibrit	2,235399	2,164564	2,105599

Tablo 4.10 ve Tablo 4.11 birlikte incelendiğinde, iterasyon bazlı ortalamaların ortalamasına bakılırsa 64 boyut için CUDA 3,426886, hibrit 1,719652 hızlanma üretmiştir. 128 boyut için ise CUDA 3,579289, hibrit 2,168520 hızlanma değerlerini oluşturmuştur. Bu sonuçlar,

problem boyutu arttığında yani problem zorlaştığında CUDA etkisinin hızlanmayı arttırdığını göstermektedir.

5. SONUÇLAR VE DEĞERLENDİRME

Bu tez çalışmasında, PSO algoritması CPU, CUDA ve hibrit yapı kullanılarak üç farklı şekilde uygulanmıştır. 64 ve 128 boyuta sahip 18 farklı test fonksiyonu, 1000, 2500 ve 5000 iterasyon için CPU, CUDA ve hibrit yöntemler kullanılarak 64 parçacıklı PSO algoritması ile çözülmüştür. Her bir deneyin sonuçları 100 farklı bağımsız çalıştırma ile bulunmuştur. CUDA ve hibritin hızlanmaya etkisini incelemek amacıyla her bir deney için harcanan süre kaydedilmiş ve hızlanma değeri hesaplanmıştır. Elde edilen veriler incelendiğinde bazı test fonksiyonlarında daha yüksek, bazılarında ise daha düşük hızlanma görülmüştür. CUDA yönteminde 64 boyut için en yüksek 9,120834, en düşük 1,927629, ortalama 3,426886, 128 boyut için ise en yüksek 7,034139, en düşük 2,037173, ortalama 3,579289 hızlanma değerleri elde edilmiştir. Hibrit yöntem ise 64 boyut için en yüksek 3,465646, en düşük 1,087825, ortalama 1,719652, 128 boyut için ise en yüksek 7,136033, en düşük 1,046644, ortalama 2,168520 hızlanma üretmiştir. Ortalama hızlanma değerlerine bakıldığında, problem boyutu arttıkça yani problem zorlaştıkça hızlanmanın arttığı görülmektedir. CUDA ve hibrit yöntemler ortalama hızlanma değerlerine göre karşılaştırıldığında CUDA’nın problem boyutu 64 için yaklaşık 2 kat, 128 için yaklaşık 1,5 kat daha hızlı olduğu ortaya konmuştur.

Gelecek çalışmalarımızda, yeni hibrit yöntemlerin geliştirmesi ve farklı tür problemler üzerindeki hızlanma etkilerinin araştırılması planlanmaktadır.

6. KAYNAKLAR

[1] A. K. Dorgushaova, V. I. Zarubin, S. K. Kuizheva, T. A. Ovsyannikova, and E. N. Klochko, “The Control Actions Structure Optimization in the Process of the Regions Housing and Communal Services Development”, IJEFI, vol. 6, no. 1S, pp. 180–186, May 2016.

[2] I. Cholissodin and S. Sutrisno, “Prediction of rainfall using improved deep learning with particle swarm optimization,” *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 5, p. 2498, Oct. 2020, doi: 10.12928/telkomnika.v18i5.14665.

[3] F. Ayata, T. Uçkan, A. Karcı, and E. Seyyarer, “Derin Öğrenmede Kullanılan Optimizasyon Algoritmalarının Uygulanması ve Kıyaslanması,” *Anatolian Science*, pp. 90–98, 2020.

[4] P. Qu, J. Yan, Y.-H. Zhang, and G. R. Gao, “Parallel Turing Machine, a Proposal,” *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 269–285, Mar. 2017, doi: 10.1007/s11390-017-1721-3.

[5] V. Soni, A. Sharma, and V. Singh, “A Critical Review on Nature Inspired Optimization Algorithms,” *IOP Conference Series: Materials Science and Engineering*, vol. 1099, no. 1, p. 012055, Mar. 2021, doi: 10.1088/1757-899X/1099/1/012055.

[6] M. Özsağlam and M. Çunkaş, “Optimizasyon Problemlerinin Çözümü için Parçaçık Sürü Optimizasyonu Algoritması,” *Politeknik Dergisi Journal of Polytechnic Cilt*, vol. 11, no. 4, pp. 299–305, 2008, doi: 10.2339/2008.11.4.

[7] B. Barney and D. Frederick, “Introduction to Paralel Computing (LLNL),” 2021. [Online] Erişim adresi: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. Erişim tarihi: 10 Şubat 2022

[8] B. Wilkinson and M. Allen, “Parallel programming: techniques and applications using networked workstations and parallel computers,” 2nd ed. Upper Saddle River, NJ 07458: Pearson, 2004.

[9] U. Ercan, H. Akar ve A. Koçer, “Paralel Programlamada Kullanılan Temel Algoritmalar,” ss.812-816, XV. Akademik Bilişim Konferansı, Antalya, Türkiye, 01- 04 Şubat 2013.

[10] A. Masat, C. Colombo, and A. Boutonnet, "Surfing Chaotic Perturbations in Interplanetary Multi-Flyby Trajectories: Augmented Picard-Chebyshev Integration for Parallel and GPU Computing Architectures," Jan. 2022. doi: 10.2514/6.2022-1275.

[11] J. Han and B. Sharma, "*Learn Cuda Programming:* A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++," 27 September 2019 Packt Publishing.

[12] S. Nobile, P. Cazzaniga, D. Besozzi, D. Pescini, and G. Mauri, "cuTauLeaping: A GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems," *PLoS ONE*, vol. 9, no. 3, Mar. 2014, doi: 10.1371/journal.pone.0091963.

[13] NVIDIA, "CUDA Deep Neural Network (cuDNN)," 2022. [Online] Erişim adresi: https://developer.nvidia.com/cudnn. Erişim tarihi: 3 Nisan 2022.

[14] Jiri Kraus, "An Introduction to CUDA-Aware MPI," Mar. 13, 2013. [Online] Erişim adresi: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/. Erişim tarihi: 13 Mart 2022.

[15] NVIDIA, "CURAND," *NVIDIA*, May 11, 2022. [Online] Erişim adresi: https://docs.nvidia.com/cuda/curand/. Erişim tarihi: 20 Mayıs 2022.

[16] M. S. Kıran and A. C. Çınar, "Ağaç-tohum algoritmasının CUDA destekli grafik işlem birimi üzerinde paralel uygulaması," *Gazi Üniversitesi Mühendislik-Mimarlık Fakültesi Dergisi*, vol. 2018, no. 2018, Apr. 2018, doi: 10.17341/gazimmfd.416436.

[17] X. Yang, *Nature-Inspired Metaheuristic Algorithms Second Edition*, no. May. 2010. doi: 10.1016/j.chemosphere.2009.08.026.

[18] M. Cirnu and I. Badralexi, "On Newton-Raphson Method," *Journal of Information Systems and Operations Management*, vol. 5, pp. 91–94, 2011.

[19] I. Moser, "Hooke-Jeeves revisited," *2009 IEEE Congress on Evolutionary Computation, CEC 2009*, no. June 2009, pp. 2670–2676, 2009, doi: 10.1109/CEC.2009.4983277.

[20] J. Mathews and K. K. Fink, "Nelder-Mead method," *Numerical Methods Using Matlab*, pp. 430–437, 2004.

[21] X. Yang, "Nature-Inspired Metaheuristic Algorithms: Second Edition," Luniver Press July 2010.

[22] M. Owen, "G. Brown and G. Yule, Discourse analysis. Cambridge: Cambridge University Press, 1983. Pp. xii + 288. - M. Stubbs, Discourse analysis. Oxford: Basil Blackwell, 1983. Pp. xiv + 272.," *Journal of Linguistics*, vol. 21, no. 1, pp. 241–245, 1985, doi: 10.1017/s0022226700010161.

[23] J. H. Holland, “Genetic Algorithms-John H. Holland,” pp. 1–4, 1975, [Online] Erişim Adresi: http://www.econ.iastate.edu/tesfatsi/holland.GAIntro.htm. Erişim tarihi: 22 Mart 2022

[24] T. Bäck and H.-P. Schwefel, “An Overview of Evolutionary Algorithms for Parameter Optimization,” *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, Mar. 1993, doi: 10.1162/evco.1993.1.1.1.

[25] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

[26] P. Wongthongtham, T. Dillon, K. Tochtermann, and S. Eds, “Harmony Search Algorithms for Structural Design Optimization Studies in Computational Intelligence , Volume 239,” *Distributed Computing*.

[27] C. Tovey, “The Honey Bee Algorithm: A Biological Inspired Approach to Internet Server Optimization,” *Engineering Enterprise*, no. January 2004, pp. 13–15, 2004.

[28] D. Karaboga and B. Basturk, “Artificial Bee Colony (ABC) Optimization Algorithm for Solving Constrained Optimization Problems,” in *Foundations of Fuzzy Logic and Soft Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 789–798. doi: 10.1007/978-3-540-72950-1_77.

[29] L. Mussi and S. Cagnoni, “Particle Swarm Optimization within the CUDA Architecture,” *Proceedings of the 11th Annual conference on Genetic and evolutionary computation- GECCO ’09*, no. January 2009, pp. 1–2, 2009.

[30] D. Datta, S. Mehta, Shalivahan, and R. Srivastava, “CUDA based Particle Swarm Optimization for geophysical inversion,” in *2012 1st International Conference on Recent Advances in Information Technology (RAIT)*, Mar. 2012, pp. 416–420. doi: 10.1109/RAIT.2012.6194456.

[31] S. Solomon, P. Thulasiraman, and R. K. Thulasiram, “Collaborative multi-swarm PSO for task matching using graphics processing units,” *Genetic and Evolutionary Computation Conference, GECCO’11*, pp. 1563–1570, 2011, doi: 10.1145/2001576.2001787.

[32] J. Platos, V. Snasel, T. Jezowicz, P. Kromer, and A. Abraham, “A PSO-based document classification algorithm accelerated by the CUDA Platform,” *Conference Proceedings-IEEE International Conference on Systems, Man and Cybernetics*, pp. 1936–1941, 2012, doi: 10.1109/ICSMC.2012.6378021.

[33] G. H. Luo, S. K. Huang, Y. S. Chang, and S. M. Yuan, “A parallel Bees Algorithm implementation on GPU,” *Journal of Systems Architecture*, vol. 60, no. 3, pp. 271–279, 2014, doi: 10.1016/j.sysarc.2013.09.007.

[34] J. Janousešek, P. Gajdoš, M. Radecký, and V. Snášel, “Classification via Nearest Prototype Classifier Utilizing Artificial Bee Colony on CUDA,” 2014, pp. 21–30. doi: 10.1007/978-3-319-07995-0_3.

[35] P. Wang, H. Li, and B. Zhang, “A GPU-based parallel ant colony algorithm for scientific workflow scheduling,” *International Journal of Grid and Distributed Computing*, vol. 8, no. 4, pp. 37–46, 2015, doi: 10.14257/ijgdc.2015.8.4.04.

[36] K. Zhang, M. Qiu, L. Li, and X. Liu, “Accelerating genetic algorithm for solving graph coloring problem based on CUDA architecture,” *Communications in Computer and Information Science*, vol. 472, pp. 578–584, 2014, doi: 10.1007/978-3-662-45049-9_95.

[37] E. H. M. Silva and C. J. A. Bastos Filho, “PSO Efficient Implementation on GPUs Using Low Latency Memory,” *IEEE Latin America Transactions*, vol. 13, no. 5, pp. 1619–1624, May 2015, doi: 10.1109/TLA.2015.7112023.

[38] A. Zarrabi, E. K. Karuppiah, Y. K. Kok, N. C. Hai, and S. See, “Gravitational Search Algorithm Using CUDA,” *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, vol. 2015-July, pp. 193–198, 2015, doi: 10.1109/PDCAT.2014.38.

[39] A. Ouyang, Z. Tang, X. Zhou, Y. Xu, G. Pan, and K. Li, “Parallel hybrid PSO with CUDA for lD heat conduction equation,” *Computers and Fluids*, vol. 110, pp. 198–210, 2015, doi: 10.1016/j.compfluid.2014.05.020.

[40] L. Bukata, P. Šůcha, and Z. Hanzálek, “Solving the resource constrained project scheduling problem using the parallel Tabu Search designed for the CUDA platform,” *Journal of Parallel and Distributed Computing*, vol. 77, pp. 58–68, 2015, doi: 10.1016/j.jpdc.2014.11.005.

[41] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95- International Conference on Neural Networks*, Dec. 1995, pp. 1942–1948. doi: 10.1109/ICNN.1995.488968.

[42] Axel Thevenot, “78 Benchmark Test Functions for Single Objective Optimization,” *TowardsDataScience*, Dec. 31, 2020. [Online] Erişim adresi: https://towardsdatascience.com/optimization-eye-pleasure-78-benchmark-test-functions-for-single-objective-optimization-92e7ed1d1f12. Erişim tarihi: 5 Ekim 2021

[43] S. Özyön, C. Yaşar, and H. Temurtaş, “Incremental gravitational search algorithm for high-dimensional benchmark functions,” *Neural Computing and Applications*, vol. 31, no. 8, pp. 3779–3803, Aug. 2019, doi: 10.1007/s00521-017-3334-8.
[44] Agustinus Kristiadi, “Comparison between CUDA and CPU for PSO,” Nov. 10, 2012. [Online] Erişim adresi: https://github.com/wiseodd/cuda-pso. Erişim tarihi: 15 Eylül 2021.
[45] Chintan Soni, “About A GPU Accelerated SPSO Implementation in CUDA,” Dec. 08, 2019. [Online] Erişim adresi: https://github.com/c-soni/SPSO. Erişim tarihi: 22 Eylül 2021.

ÖZGEÇMİŞ

Kişisel Bilgiler

Adı Soyadı : Muhammet Taha Aydın

Doğum tarihi ve yeri :

e-posta :

Öğrenim Bilgileri

Derece	Okul/Program	Yıl
Y. Lisans	Balıkesir Üniversitesi/Elektrik-Elektronik Mühendisliği	-
Lisans	Karabük Üniversitesi/Bilgisayar Mühendisliği	2017
Lise	Gülser Mehmet Bolluk Anadolu Lisesi	2012