İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DISTRIBUTED OPENCL
OPENCL PLATFORMUNUN AĞ ÖLÇEĞİNDE DAĞITILMASI

YÜKSEK LİSANS TEZİ

Barış Eskikaya

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Yüksek Lisans Programı

Tez Danışmanı: Yrd. Doç. Dr. Deniz Turgay ALTILAR

Haziran 2012

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DISTRIBUTED OPENCL
OPENCL PLATFORMUNUN AĞ ÖLÇEĞİNDE DAĞITILMASI

YÜKSEK LİSANS TEZİ

Barış Eskikaya
(504091505)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Yüksek Lisans Programı

Tez Danışmanı: Yrd. Doç. Dr. Deniz Turgay ALTILAR

Haziran 2012

İTÜ, Fen Bilimleri Enstitüsü'nün 504091505 numaralı Yüksek Lisans / Doktora Öğrencisi **Barış ESKİKAYA**, ilgili yönetmeliklerin belirldiği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"DISTRIBUTED OPENCL - OPENCL PLATFORMUNUN AĞ ÖLÇEĞİNDE DAĞITILMASI"** başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı : **Yrd. Doç. Dr. Deniz Turgay ALTILAR** ....
İstanbul Teknik Üniversitesi

Jüri Üyeleri : **Yrd. Doç. Dr. Deniz Turgay ALTILAR** ....
İstanbul Teknik Üniversitesi

Prof. Dr. Nadia ERDOĞAN ....
İstanbul Teknik Üniversitesi

Prof. Dr. Can ÖZTURAN ....
Boğaziçi Üniversitesi

Teslim Tarihi : **18 Temmuz 2012**
Savunma Tarihi : **18 Haziran 2012**

ÖNSÖZ

Yüksek lisans çalışmam boyunca bilgi, birikim ve desteğini benden esirgemeyen değerli hocam Yrd. Doç. Dr. Deniz Turgay Altılar’a teşekkürü borç bilirim.

Haziran 2012

Barış Eskikaya
Bilgisayar Mühendisi

İÇİNDEKİLER

Sayfa

KISALTMALAR

API : Application Programming Interface
CPU : Central Processing Unit
DCL : Distributed OpenCL
GFLOPS/s : Giga Floating-Point Operations per Second
GPU : Graphic Processing Unit
GPGPU : General Purpose programming on Graphic Processing Unit
HPC : High Performance Computing
JSON : JavaScript Object Notation
LAN : Local Area Network
Mbps : Megabits per second
NCL : Native OpenCL
RPC : Remote Procedure Call
SIMD : Single Instruction Multiple Data
SPMD : Single Program Multiple Data

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

DISTRIBUTED OPENCL - OPENCL PLATFORMUNUN AĞ ÖLÇEĞİNDE DAĞITILMASI

ÖZET

Grafik kartları, yapısındaki çok çekirdeklilik ve işlem hattı mimarisi sayesinde yüksek miktardaki bağımlı olmayan verileri paralel olarak hızlı bir şekilde işleyebilme yeteneğine sahiptir. Ekran kartlarının bu özelliğinden bilimsel hesaplamalar, benzetim uygulamaları, işaret işleme uygulamaları gibi alanlarda da yararlanabilmek amacıyla grafik işlem birimi üzerinde genel amaçlı programlama (GPGPU) kavramı ortaya çıkmıştır.

OpenCL, son zamanlarda yaygın olarak kullanılmaya başlayan bir GPGPU çatısıdır. OpenCL, çeşitli üreticilere ait çeşitli modellerdeki CPU ve GPU aygıtlarının bulunduğu heterojen ortamlarda çalışan ve aygıtlar arasında taşınabilir programlar yazmayı mümkün kılmaktadır. Üreticiler tarafından kabul edilen ortak OpenCL standartları sayesinde hem programcılar aygıtlara özel teknik detaylardan soyutlanmakta hem de platform ve aygıt özelliklerinden bağımsız programlar yazılabilmektedir. Bu nedenle OpenCL, önde gelen üreticiler tarafından desteklenen ve giderek daha yaygın olarak kullanılan bir GPGPU çatısı haline gelmektedir.

Bu tez çalışmasında; tek bir bilgisayarın kaynaklarını kullanarak çalışan OpenCL çatısını, JSON-RPC iletişim tekniği kullanarak ağ ölçeğinde dağıtılmış olan birçok bilgisayardaki kaynakları paralel olarak kullanabilecek hale getiren Distributed OpenCL (Dağıtılmış OpenCL) çatısı geliştirilmiştir. Geliştirilen bu yeni yapı ile OpenCL çatısının birçok bilgisayardaki GPU veya CPU kaynaklarının aynı işlem üzerinde paralel olarak çalışıp aygıt başına düşen işlem yükünü azaltarak genel hesaplama performansını arttıracak ve aynı zamanda aygıt, üretici ve işletim sisteminden bağımsız olarak çalışacak şekilde genişletilmesi amaçlanmıştır.

Geliştirilen yapı istemci – sunucu mimarisinde çalışan bir sistemdir. İstemci, dağıtım katmanı ve sunucu bileşenlerinden oluşmaktadır. İstemcilerde çalışan ev sahibi programların yaptığı OpenCL API çağrıları yine istemcide çalışan dağıtım katmanı tarafından yakalanmakta, OpenCL çağrıları, çağrılara ait parametrelerle birlikte

JSON mesajlarına çevrilmekte ve sistemdeki sunuculara JSON – RPC protokolü kullanılarak iletilmektedir. Sunucu, kendisine gelen mesajları dinlemekte, bir mesaj geldiğinde JSON formatındaki mesaj içeriğini ayıklayarak ilgili OpenCL çağrısını verilen parametrelere birlikte işletmekte, dönen sonuç parametrelerini yine JSON mesajı haline getirerek istemciye cevap olarak iletmektedir. Mesaj cevabını alan istemcideki dağıtım katmanı, dönüş değerlerini OpenCL çağrısının işlevini göz önünde bulundurarak ev sahibi programa döndürmektedir.

Geliştirilen bu yeni yapı üzerinde Doğal OpenCL (Native OpenCL) ile karşılaştırmalı olarak bazı testler yapılmıştır. Bu testler ek yük testleri, paralel yürütme testleri ve çekirdek döngüsü testleri olarak sınıflandırılmaktadır. Ek yük testlerinde Native OpenCL ile Distributed OpenCL'de sunucu ve istemcinin aynı bilgisayarda çalışması, sunucu ve işlemcinin farklı bilgisayarlarda çalışması durumlarındaki performans karşılaştırılmıştır. Paralel yük testlerinde Distributed OpenCL'in bir istemci – bir sunucu, bir istemci – iki sunucu, bir istemci – dört sunucu şeklinde çalışması gözlemlenmiştir. Çekirdek döngüsü testlerinde hesap yoğun bir çekirdek fonksiyonu benzetimi yapılarak bu durumda sistemler arasındaki performans farkı ölçülmüştür.

Testler sonucunda Distributed OpenCL çatısının özellikle düşük gecikmeli – yüksek hızlı ağlar ve çoklu GPU barındırabilen aygıtlar ile birlikte kullanıldığında; üretici – işletim sistemi bağımsız, dağıtık, paralel çalışan ve ölçeklenebilir yapıya sahip bir GPGPU hesaplama ortamı sağlayabileceği görülmüştür.

DISTRIBUTED OPENCL - DISTRIBUTING OPENCL PLATFORM ON NETWORK SCALE

SUMMARY

Graphics cards have the ability to process with high performance on massive amounts of independent data in parallel as a result of their multi-core and pipeline architecture. To use these advantages of GPU in other fields such as scientific calculations, simulation applications and signal processing applications GPGPU (General Purpose programming on GPU) concept has come into use.

OpenCL is a GPGPU framework which is becoming more popular day by day. OpenCL provides developing GPGPU applications which execute in heterogen environments consisting of GPU, CPU and other processors. OpenCL programs are also portable among devices. Many producers accept and support OpenCL standards. OpenCL also abstracts technical details of vendor specific implementations and devices architectures from developers.

The features described above make OpenCL a reliable and widespread use GPGPU framework. Considering the advantages and properties of OpenCL, it is possible to extend the benefits of OpenCL by designing a system which consists of distributed computing resources and hosts that use these computing resources connected via network.

In this study Distributed OpenCL framework, which extends OpenCL on network scale using JSON RPC communication technique between hosts and computing resources, has been developed. Since the framework is based on OpenCL, it covers the advantages of OpenCL such as heterogeneous resource usage, vendor and architecture independency, moreover it increases the level of parallelism by increasing number of computing resources and by using JSON RPC for communication it enables operating system and platform independency in communication layer.

Distributed OpenCL framework has a client - server architecture that can consist of multiple clients and multiple servers. Distributed OpenCL framework consists of the following components:

- Clients
- Distribution Layer
- Servers

Clients run OpenCL host applications and OpenCL API calls from host applications are redirected to servers on the network by distribution layer running on the client machines. OpenCL functionality is provided on servers.

When the OpenCL host application running on the client makes an API call, this call is sent to the distribution layer running on the client. Distribution layer converts API calls to JSON message packages by marshalling. These JSON messages are sent to the servers registered in the system using JSON RPC communication technique. JSON RPC server application on the server machines listens to the incoming JSON messages and converts these messages to OpenCL API calls by unmarshalling. With this method, OpenCL API calls are replicated on the server with the same parameters from the client. Server converts responses from OpenCL computing devices to JSON messages and send back result to client which made the API call. Since OpenCL computing devices keep the context and state data of program in their own memory, servers do not keep any context or state data of client programs. Pointers to the instances of OpenCL types such as *cl_mem*, *cl_context*, *cl_program*, *cl_kernel* etc. that are created by OpenCL API calls are allocated in heap section of server memory to provide access by pointer addresses of them to use in following OpenCL API calls. These allocated spaces are freed when release functions such as *clReleaseMemObject*, *clReleaseContext* etc. are called by the corresponding host application.

Considering GPU architecture performs operations in SIMD model, data which will be processed can be distributed to multiple servers registered in the system. This feature reduces the workload of one server so reduces the workload of each processing unit in the server and results speedup for calculation by increasing level of parallelism.

A set of tests is performed to compare performances of Native OpenCL and Distributed OpenCL framework. These tests are classified as overhead tests, parallel execution tests and kernel loop tests. In overhead tests; performance of Native OpenCL and Distributed OpenCL configurations (client and server running on the same machine, client and server running different machines) are compared. In parallel execution tests; exetucion times for Distributed OpenCL configurations (one client – one server, one client – two servers, one client – four servers) are observed. In kernel loop tests; additon operation in the kernel function is looped to simulate computationally dense kernel functions. Performance difference between the systems is measured.

As evaluation of the tests, it is seen that Distributed OpenCL can provide a vender - operating system independent, distributed, parallel and scalable GPGPU computing environment especially when it is used with devices like multi GPU servers and low latency – high speed networks like *Infiniband*.

1. GİRİŞ

Grafik işlem birimleri (GPU), işlem hattı (pipeline) mimarisine sahip olduklarından tek komut çoklu veri (SIMD – Single Instruction Multiple Data) tipinde çalışma yapısını desteklemektedirler. Bu sayede yüklü miktarda veriyi paralel olarak işleyerek veri seviyesinde paralellik (data-level parallelism) sağlayabilirler. İlkel GPU'lar sadece grafik gösterimi için kullanılbilmekte ve programlanabilir arayüzlere sahip değillerdir. Fakat GPU teknolojisindeki ilerlemelere birlikte günümüzde kullanılan modern GPU'lar programlanabilir hale gelmiş ve genel amaçlı GPU programlama (GPGPU – General Purpose programming on GPU) kavramı ortaya çıkmıştır. Bu kavramın ortaya çıkması ve gelişmesi sonucunda GPU'lar yüksek seviyeli programlama dilleriyle progamlanabilir hale gelmiş ve sadece grafik işlemleri için değil aynı zamanda bilimsel hesaplamalar gibi yüklü miktarda veri üzerinde yoğun hesaplamalı işlemlerin paralel olarak hızlı bir biçimde yapılmasına ihtiyaç duyan sistemlerde de kullanılmaya başlanmıştır. Günümüzde GPU'lar merkezi işlem birimlerinin (CPU – Central Processing Unit) yanı sıra bilgisayarların hesaplama kaynakları olarak kullanılmaktadır.

GPGPU kavramının yaygın olarak kullanılılmaya başlanmasıyla birlikte farklı üreticiler zaman içerisinde C for Graphics, Close to Metal (AMD / ATI), CUDA (NVIDIA) [1], BrookGPU (Stanford Üniversitesi) [2], DirectCompute (Microsoft) [3] gibi GPGPU programlama dilleri geliştirmişlerdir. Fakat bu programlama dilleri üretici marka veya aygıt modellerine özel olmaları nedeniyle heterojen aygıtların bulunduğu ortamlarda kullanılamamaktadırlar. Ayrıca bu programlama dillerinin özellikleri gereği, bu diller ile programlama yapan geliştiricilerin dile özel veya programladıkları aygıt mimarisine özel detaylara hâkim olmaları gerekmektedir.

Bu problemlerin üstesinden gelmek için Apple, IBM, Intel, AMD ve NVIDIA gibi ana üreticilerin katkısı ve işbirliği ile kar amacı gütmeyen bir proje olarak Khoronos Group tarafından OpenCL çatısı (OpenCL framework) [4] geliştirilmiştir. OpenCL çatısı, C tabanlı programlama dili ve genel geçer bir uygulama programlama arayüzü (API – Application Programming Interface) barındırmaktadır. OpenCL çatısı ile programcıların çeşitli üreticilere ait, çeşitli modellerdeki CPU ve GPU aygıtlarının bulunduğu heterojen ortamlarda çalışabilir ve taşınabilir programlar yazmaları mümkün kılınmıştır. Her üretici OpenCL standartlarının uygulamasını kendine özel yapmasına rağmen, OpenCL veri tipleri ve API fonksiyonları imzalarında (method signatures) ortak bir standart sağlandığı için OpenCL çatısı ile gerçeklenen programlarlar üretici ve aygıt bağımsız çalışabilmektedir. Bu ortak standartlar sayesinde OpenCL, geliştiricileri üreticiye özel veya aygıta özel teknik detaylardan da soyutlamaktadır. Bu nedenlerden dolayı OpenCL birçok üretici tarafından desteklenen ve giderek daha yaygın olarak kullanılan bir GPGPU çatısı haline gelmektedir.

Bu tez çalışmasında; tek bir bilgisayarın kaynaklarını kullanarak çalışan OpenCL çatısını, JSON-RPC [5] iletişim tekniği kullanarak ağ ölçeğinde dağıtılmış olan birçok bilgisayardaki kaynakları paralel olarak kullanabilecek hale getiren Distributed OpenCL (Dağıtılmış OpenCL) çatısı geliştirilmiştir. Geliştirilen bu yeni yapı ile OpenCL çatısının birçok bilgisayardaki GPU veya CPU kaynaklarının aynı işlem üzerinde paralel olarak çalışıp aygıt başına düşen işlem yükünü azaltarak genel hesaplama performansını arttıracak ve aynı zamanda aygıt, üretici ve işletim sisteminden bağımsız olarak çalışacak şekilde genişletilmesi amaçlanmıştır.

Bölüm 2'de grafik kartlarının ve grafik işlem birimlerinin mimarisi ile GPGPU kavramı genel özellikleriyle anlatılmıştır. Bölüm 3'te OpenCL çatısının mimarisi ve özellikleri; Bölüm 4'te JSON-RPC teknolojisi ve özellikleri, Bölüm 5'te geliştirilen yeni çatının özellikleri, nasıl gerçeklendiği, yapılan testler, test sonuçları ve önceki çalışmalarla karşılaştırmalar açıklanmıştır. Bölüm 6'da tez çalışmasından elde edilen sonuçlar değerlendirilmiştir.

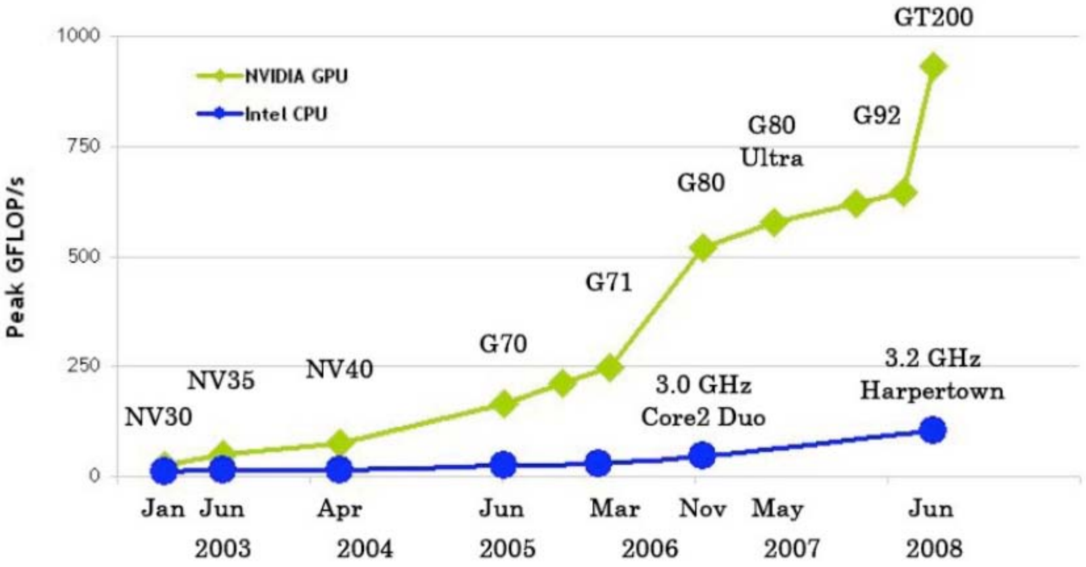
2. GRAFİK KARTLARI, GRAFİK İŞLEM BİRİMLERİ VE GPGPU KAVRAMI

2.1 Grafik Kartları

Grafik kartları, bilgisayarlarda grafik ile ilgili işlemleri yüksek performanslı ve verimli bir şekilde gerçekleştirmek ve görüntüleme birimine çıktı üretmek için özelleşmiş kartlardır. Grafik kartları; aygıtın bilgisayar donanımıyla etkileşimini gerçekleştirn alt seviyedeki gömülü yazılımının bulunduğu bios, grafik kartı belleği, hesaplama ve komut yürütme işlemlerini gerçekleştiren grafik işlem birimi (GPU) kısımlarından oluşur. Grafik kartlarını merkezi işlem birimlerinden (CPU) farklı kılan temel özellik GPU'nun tek komut çoklu veri (Single Instruction Multiple Data - SIMD) yapısında çalışarak yüklü miktarda veriyi paralel bir şekilde işleyebilmesini sağlayan mimarisidir.

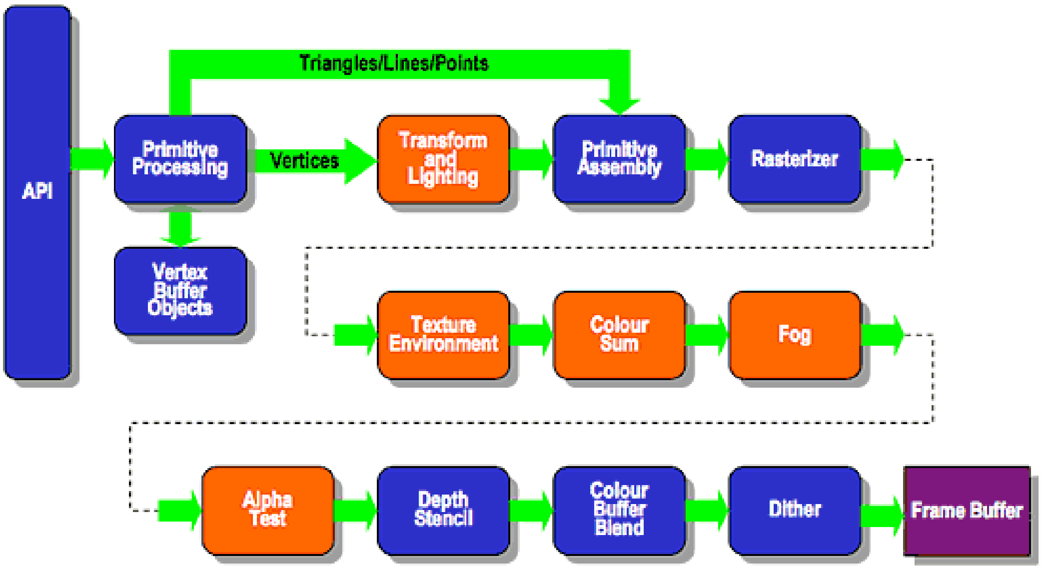
2.2 Grafik İşlem Birimi Mimarisi

Grafik İşlem Birimleri; grafik uygulamalarındaki 3 boyutlu verinin gösterim için 2 boyuta dönüştürülmesi (3D to 2D transformation), düğümlerin birleştirilmesi (vertex assembly), parça oluşturma (fragmentation), parçaların piksellere dönüştürülmesi, doku haritalama (texture mapping), gürültü giderme ve pürüzsüzleştirme (anti-aliasing) gibi çok miktarda veri üzerinde yoğun hesap gerektiren matris ve vektör işlemlerini yüksek hızda gerçekleştirebilmek üzere tasarlanmışlardır. Bu yüzden grafik işlem birimleri, merkezi işlem birimlerine göre çok daha fazla sayıda çekirdek ve işlem hattına sahiptirler. Günümüzde ev kullanıcıları tarafından kullanılan CPU'larda 2 ila 8 çekirdek bulunmaktayken GPU'larda çekirdek sayısı 80 ila 100 civarında olabilmektedir. Şekil 2.1'de Intel CPU'ların ve NVIDIA GPU'ların GFLOPS/s olarak işlem hızı bakımından 2003 ila 2008 yılları arası gelişimi gösterilmiştir. Şekilde görüleceği üzere günümüzde GPU'lar işlem hızı bakımından CPU'lara göre çok daha ileridedir. Bu farkı sağlayan temel özellik, GPU'ların genellikle paralel olarak işlenebilir veriler üzerinde çalışması ve SIMD yapısında çalışabilen gelişmiş ve karmaşık işlem hattı mimarilerine sahip olmalarıdır.

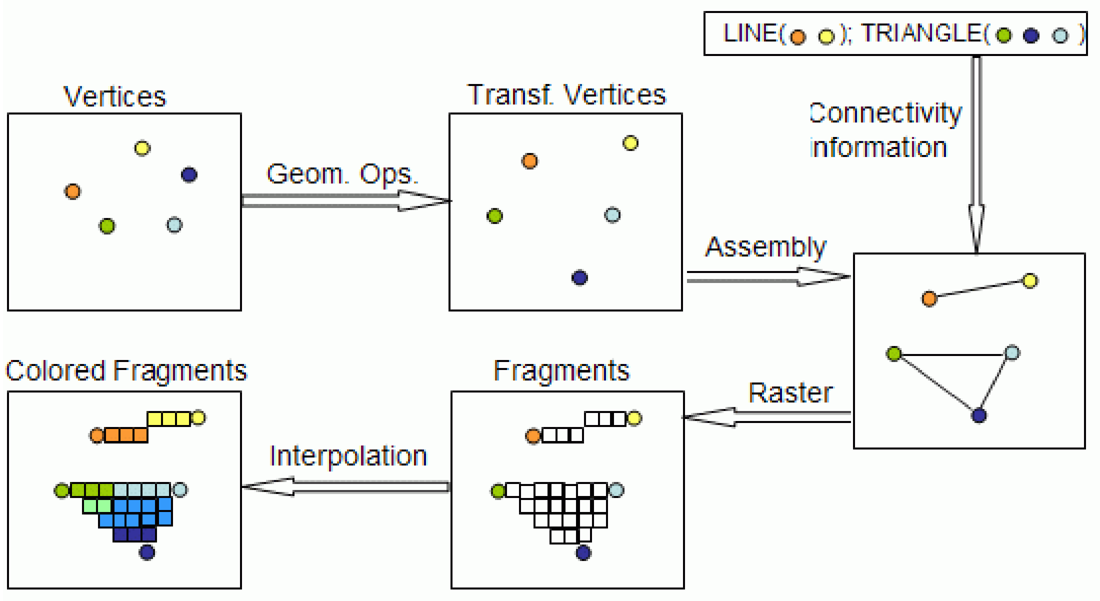


Şekil 2.1 : Intel CPU ve NVIDIA GPU'ların işlem hızı karşılaştırması [6].

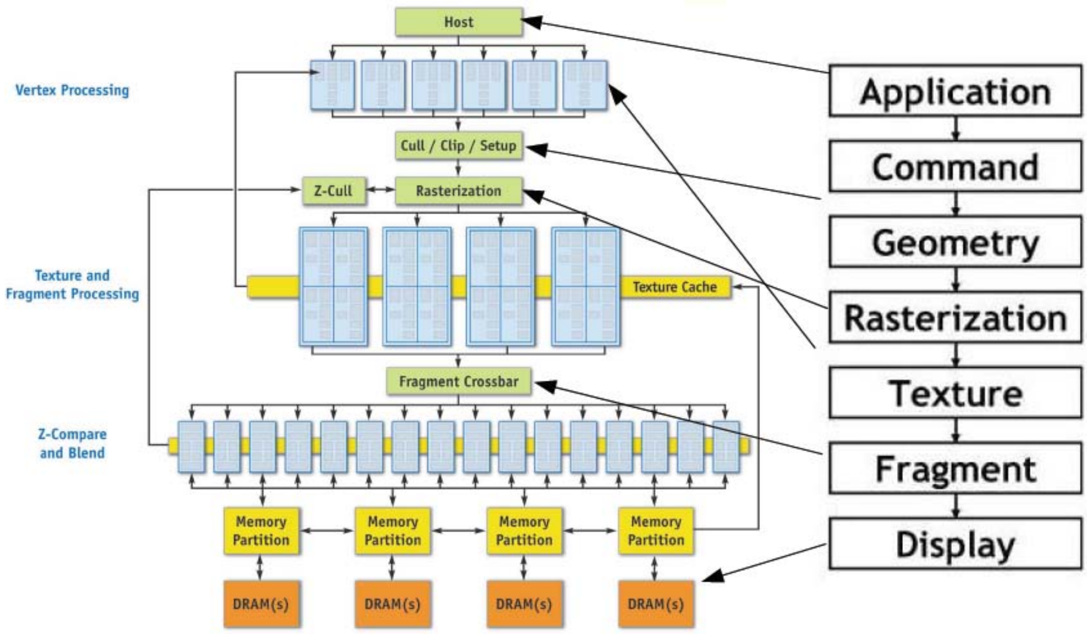
İşlem hattı birbiri ardına gerçekleşen çeşitli işlem aşamalarından oluşur. Her aşamada bir önceki aşamanın sonucu girdi olarak alınır ve işlenir. İşlem hattı sürecinde köşe kenar dönüşümü, ışıklandırma, birleştirme, piksellere dönüştürme, doku oluşturma, renk karışımları, transparanlık, derinlik katma ve gölgelendirme gibi işlemler yapıldıktan sonra oluşan veri, görüntüleme aygıtlarına aktarılır. İşlem hattında aşamaların birbiri ardına uygulanması esnasında, bir aşamada işlenen veri bloğu bir sonraki aşamaya aktarıldığı anda aynı aşamaya, işlenmesi için yeni bir veri bloğu girer. Bu şekilde aşamaların paralel işlemeye uygun olan veriler üzerinde paralel olarak çalışması sağlanmış olur. Bu işlemler sonucunda grafik uygulamaları tarafından 3 boyutlu uzayda köşe, kenar ve renk bilgileri olarak aktarılan veri, görüntüleme aygıtı tarafından 2 boyutlu olarak görüntülenebilecek resimler haline getirilmiş olur. Şekil 2.2'de GPU işlem hattı mimarisi, Şekil 2.3'te işlem hattına giren verinin ne şekilde görüntüye dönüştürüldüğü, Şekil 2.4'te ise NVIDIA GeForce 6800 ekran kartına ait blok diyagramı üzerinde işlem hattının gerçeklenmesi gösterilmiştir. Şekil 2.2 ve Şekil 2.3'te görüleceği üzere, grafik uygulaması tarafından grafik kartı API'si kullanılarak GPU'ya 3 boyutlu düzlemde düğüm koordinatları, renkleri ve düğümler arasındaki bağlılık ilişkileri iletilir. İlk aşamada düğüm koordinatları 3 boyutlu düzelemden 2 boyutlu düzleme dönüştürülür. Dönüşüm işleminden sonra düğümler arası bağlılık bilgileri kullanılarak düğümlerden düzlemler veya çizgiler oluşturulur. Daha sonra oluşturulan bu nesneler görüntülenecek çözünürlüğe bağlı olarak görüntü matrisi üzerinde piksel parçalarına dönüştürülür. Piksel parçaları içerisinde kalan renksiz alanlar, düğümlerin renk bilgileri ve renk ağırlıkları kullanılarak renklendirilir. Renklendirme işlemi sonrasında gürültü ve pürüz azaltma, ışıklandırma, gölgelendirme işlemleri de uygulanarak daha gerçekçi bir görüntü elde edilir.



Şekil 2.2 : GPU işlem hattı mimarisi [7].



Şekil 2.3 : İşlem hattında veri dönüşümü [8].

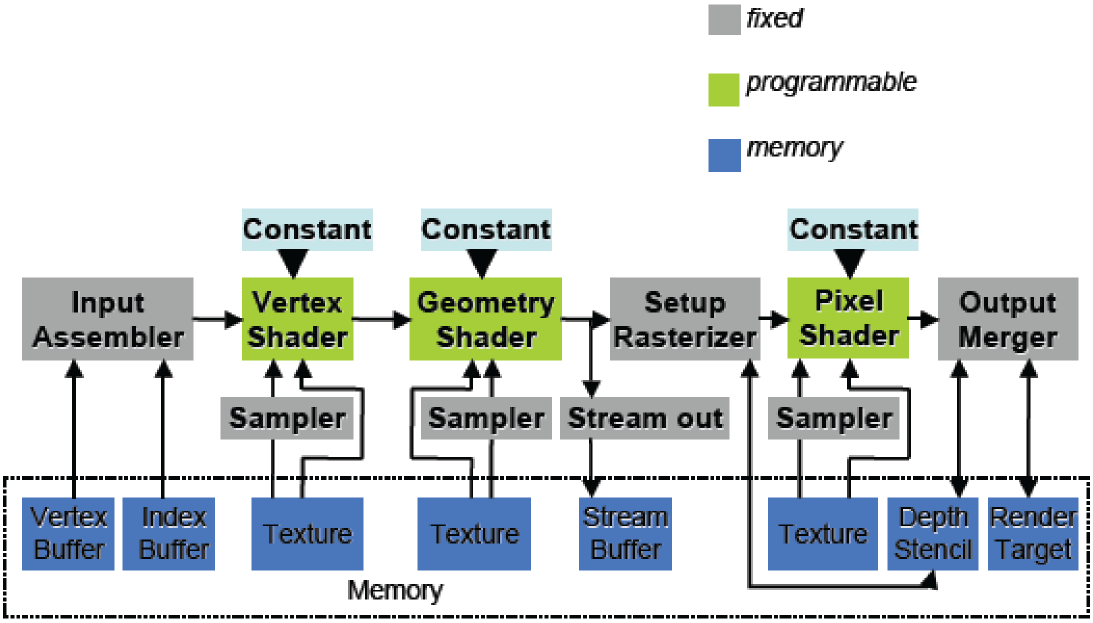


Şekil 2.4 : NVIDIA GeForce 6800 blok diyagramı [9].

2.3 GPGPU Kavramı

GPU teknolojisindeki ilerlemelerle birlikte, günümüzde kullanılan modern GPU'lar programlanabilir arayüzler sunar hale gelmişlerdir. Bu programlanabilir arayüzler sayesinde GPU'nun işlem gücü ve paralel işleyebilme yeteneği sadece grafikle ilgili uygulamalarda değil aynı zamanda genel amaçlı hesaplamalar için de kullanılabilir hale gelmiştir. Bu durum ortaya grafik işlem birimi üzerinde genel amaçlı hesaplama (GPGPU - General Purpose programming on Graphic Processing Unit) kavramını çıkarmıştır. Şekil 2.5'te programlanabilir DirectX 10 işlem hattı gösterilmektedir.

GPU'lar yukarıdaki kısımlarda açıklanan işlem hattı mimarisi sayesinde, paralel olarak işlenebilecek nitelikte verinin yüksek performanslı bir şekilde paralel olarak işlenmesi konusunda çok elverişlidirler. GPGPU uygulamaları GPU'ların grafik aygıtlarına özel olan köşe kenar dönüşümü, dokulandırma, renklendirme, gölgelendirme vb. özelliklerinden ziyade SIMD şeklinde çalışan işlem hattı mimarisinden yararlanırlar. GPGPU uygulamaları genel olarak; işaret işleme, ses işleme, görüntü işleme, şifreleme, bioinformatik, yapay sinir ağları, paralelleştirilebilen bilimsel hesaplamalar, istatiksel hesaplamalar gibi yüklü miktarda verinin küçük parçaları üzerinde bağımsız ve paralel olarak işlem yapılmasına uygun olan uygulama alanlarında başarılıdırlar.

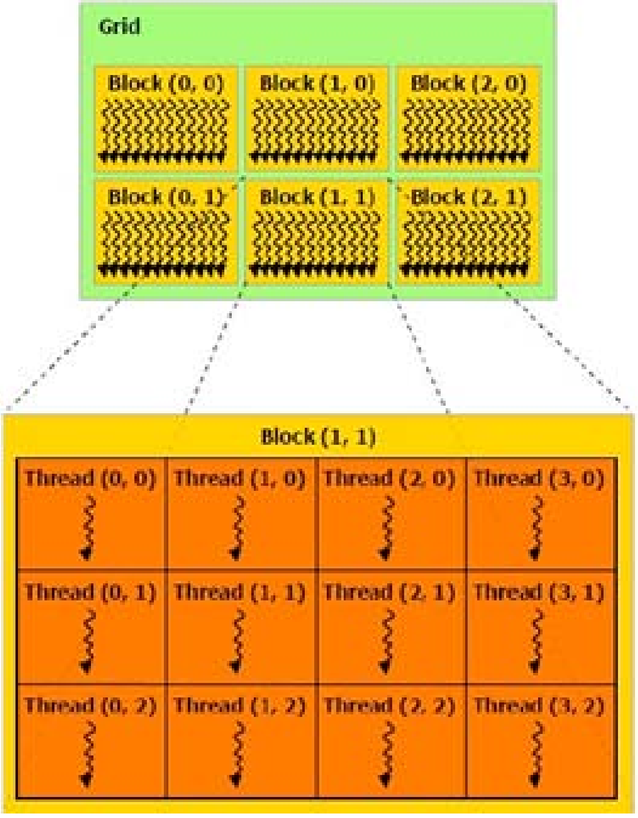


Şekil 2.5 : DirectX 10 programranabilir işlem hattı mimarisi [10].

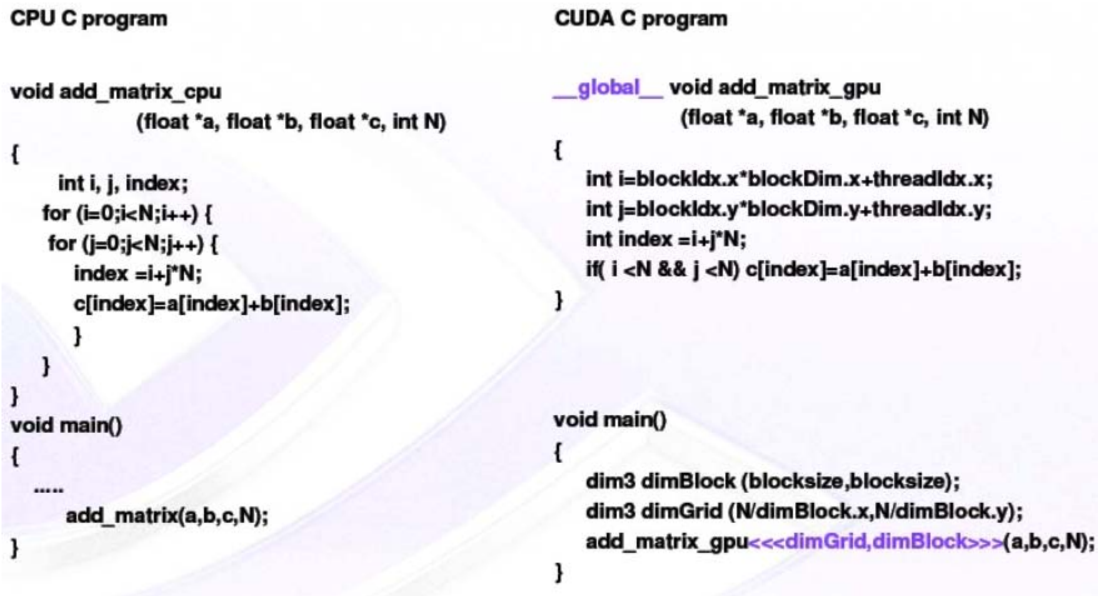
2.4 GPGPU Programlama Modeli

GPGPU uygulamaları genel olarak CPU üzerinde çalışan bir ev sahibi program (host program) ve GPU'daki çerikdekler üzerinde hesaplama yapacak olaran çekirdekk fonksiyonundan (kernel function) oluşur. Her çekirdekte çalışan çerkirdek fonksiyonu, akış (stream) şekilde GPU'ya iletilen verinin kendine düşen daha küçük bir birimi üzerinde işlem yapar. Giriş verisinin GPU'ya iletilmesi, sonuç verisinin toplanması istenilen formata dönüştürülmesi gibi ardışık işlemleri CPU'da çalışan ev sahibi program yürütür.

Şekil 2.6'da modern GPGPU dillerinden CUDA programlama diline ait programlama modeli yapısı, Şekil 2.7'de ise C programlama diliyle yazılmış CPU üzerinde çalışan bir matris toplama fonksiyonu ile yine aynı matris toplama fonksiyonunu GPU üzerinde gerçekleyen, CUDA programlama diliyle yazılmış GPU üzerinde çalışan bir kernel fonksiyonu ve C'de yazılmış bir ev sahibi program gösterilmiştir. Şekil 2.6'da görüldüğü üzere, CUDA programlama modelinde GPU aygıtı grid olarak görülür ve çok sayıda bloktan oluşur. GPU'da bulunan çok sayıdaki çekirdekten her biri aynı anda bir blok işleyebilir. Bir blok içerisinde paralel olarak çalıştırılabilen çok sayıda iplik (thread) bulunur. Bu ipliklerinin her biri kendisine düşen veri öbeği üzerinde tanımlanmış olan çekirdek fonksiyonu çalıştırır.



Şekil 2.6 : CUDA dili programlama modeli yapısı [11].



CPU C program

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

CUDA C program

```
__global__ void add_matrix_gpu
             (float *a, float *b, float *c, int N)
{
   int i=blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   int index =i+j*N;
   if( i <N && j <N) c[index]=a[index]+b[index];
}


void main()
{
   dim3 dimBlock (blocksize,blocksize);
   dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

Şekil 2.7 : Matris toplama işleminin C'de ve CUDA'da gerçeklenmesi [12].

Şekil 2.7'de görüldüğü gibi C programlama dilinde yazılmış olan matris toplama fonksiyonu matris elemanları üzerinde ardışıl döngüler şeklinde işlem yaparak matris toplama işlemini gerçekleştirir. CUDA ile yazılmış matris toplama programına bakıldığında, program CPU üzerinde çalışan ev sahibi programdaki *main* fonksiyonundan ve GPU üzerinde çalışan *add_matrix_gpu* çekirdek fonksiyonundan oluşur. Ev sahibi program bir bloğun boyutunu ve *grid* içerisindeki blok sayısını belirler. Daha sonra bloklar üzerinde paralel olarak çalışacak olan *add_matrix_gpu* fonksiyonunu çağırır. Çağrı sonucu *add_matrix_gpu* çekirdek fonksiyonu bloklar ve bloklardaki iplikler üzerinde paralel olarak yürütülür. Her bir iplik kendi iplik numarası (thread ID), blok numarası (block ID) ve blok boyutunu (blockDim) kullanarak kendine düşen veri parçası için matris toplama işlemini gerçekleştirir. İplik numarası, blok numarası, blok boyutu gibi veriler bağlam (*context*) içerisinde her bir çekirdek fonksiyona geçer.

2.5 GPGPU Programlama Dilleri

GPGPU kavramının yaygınlaşmasıyla birlikte daha önceden *assembly* programlama dilleriyle programlanan GPU'lar için yüksek seviyeli GPGPU programlama dilleri geliştirilmiştir. Yüksek seviyeli programlama dillerinin geliştiriciler tarafından anlaşılması ve yazılması assembly dillerine göre daha kolaydır. Yüksek seviyeli programlama dillerinde geliştirilen kodlar aygıtın teknik detaylarını geliştiricilerden soyutlayabilme ve aygıtlar arası taşınabilme açısından da daha avantajlılardır. Zaman içerisinde geliştirilmiş olan programlama dillerinin başlıcaları şunlardır: C for Graphics, Close to Metal, BrookGPU, CUDA, DirectCompute, OpenCL.

2.5.1 C for Graphics

C for Graphics (Cg) [13], NVIDIA ve Microsoft tarafından geliştirilmiş, köşe ve piksel parçacık işlemcilerini (vertext and pixel shaders) programlamaya yarayan C tabanlı yüksek seviyeli bir GPGPU dilidir. Cg dili ile yazılmış çekirdek fonksiyonları OpenGL ve DirectX API'leri ile çağırılabilmektedir. CG programlama dili ile yazılmış çekirdek fonksiyon kaynak kodları çalışma zamanı esnasında derlenebilir.

2.5.2 Close to Metal

Close to Metal [14], ATI tafafından AMD GPU'larda kullanılmak üzere geliştirilmiş bir düşük seviyeli GPGPU dilidir. Programcılara GPU aygıtının komut setine ve belleğine doğrudan erişim sağlamaktadır. AMD kartlarda daha sonradan daha yüksek seviyeli olan Stream SDK teknolojisine geçilmiştir.

2.5.3 BrookGPU

Stanford Üniversitesinde geliştirilmiş olan BrookGPU [2], Brook akış programlama (Brook stream programming) dilinin AMD ve NVIDIA GPU aygıtları üzerinde çalışmak üzere uyarlanmış halidir. BrookGPU dili ev sahibi program tarafında API olarak OpenGL, DirectX veya Close to Metal API'leri ile Windows ve Linux platformlarda çalışabilmektedir.

2.5.4 CUDA

CUDA (Compute Unified Device Architecture) [1], NVIDIA tarafından GPU'larda kullanılmak üzere geliştirilmiş olam bir paralel hesaplama mimarisidir. GPU etkileşimi için hem alçak seviyeli hem de yüksek seviyeli API sunmaktadır.CUDA bir GPGPU dili olarak; ardışıl bellek erişimi, paylaşımlı bellek, GPU'dan daha hızlı veri okuma, tam sayı ve bit bazında (bitwise) işlemler için destek sağladığından dolayı avantajlıdır.

2.5.5 Direct Compute

Direct Compute [3], Microsoft tarafından Windows işletim sistemi üzerinde GPU programlamada kullanılmak üzere geliştirilmiş olan bir API'dir. DirectX 10 ve DirectX 11 destekleyen GPU'lar üzerinde çalışabilmektedir.

2.5.6 OpenCL

OpenCL (Open Computing Language) [4], CPU, GPU ve diğer işlemcilerden oluşabilen heterojen ortamlarda taşınabilir programlar yazılabilinmesi amacıyla geliştirilmiş bir çatıdır. Apple, IBM, Intel, AMD ve NVIDIA gibi büyük üreticilerin katkısı ve işbirliği ile kar amacı gütmeyen bir proje olarak Khoronos Group tarafından geliştirilmiştir. OpenCL çatısı, GPU üzerinde çalışan çekirdek fonksiyonların yazımı için C tabanlı programlama dili ve ev sahibi program tarafında çalışan genel geçer bir uygulama programlama arayüzü (API – Application Programming Interface) barındırmaktadır. OpenCL çatısı ile programcıların çeşitli üreticilere ait, çeşitli modellerdeki CPU ve GPU aygıtlarının bulunduğu heterojen ortamlarda çalışabilir ve taşınabilir programlar yazmaları mümkün kılınmıştır. Aygıt üreticileri OpenCL standartlarına uygun OpenCL gerçeklemelerini kapalı kaynaklı olarak kendilerine özel geliştirmektedirler fakat, standartlara uyularak ortak OpenCL veri tipleri ve API fonksiyonları imzaları (method signatures) kullanıldığı için OpenCL çatısı ile gerçeklenen programlarlar üretici ve aygıt bağımsız çalışabilmektedir. Bu ortak standartlar sayesinde OpenCL, geliştiricileri üreticiye özel veya aygıta özel teknik detaylardan da soyutlamaktadır. NVIDIA ve AMD'ye ait modern GPU'ların çoğu OpenCL standartlarına uygun üretilmiştir. Bu nedenlerden dolayı OpenCL birçok üretici tarafından desteklenen ve giderek daha yaygın olarak kullanılan bir GPGPU çatısı haline gelmektedir.

3. OPENCL

OpenCL (Open Computing Language), CPU, GPU ve diğer işlemcilerden oluşabilen heterojen ortamlarda taşınabilir programlar yazılabilinmesi amacıyla geliştirilmiş bir çatıdır. OpenCL standartları programcıların aygıtlar arasında taşınabilir, üretici ve aygıt bağımsız programlar yazabilmelerini sağlamak amacıyla geliştirilmiştir.

3.1 OpenCL Terimleri

OpenCL standartlarında üreticiler ve aygıtlar arası ortak standartları yakalamak amacıyla bazı terimler tanımlanmıştır. OpenCL uygulamaları; ev sahibi program (host), aygıt (device), program nesneleri (program objects), çekirdek fonksiyonları (kernel functions), çekirdek nesneleri (kernel objects) ve bellek nesneleri (memory objects) gibi bu terimler üzerine gerçeklenir

3.1.1 Aygıtlar

Bir bilgisayar sistemindeki hesaplama aygıtları, “aygıt” (device) olarak tanımlanır. Bir aygıt içerisinde bir veya birden fazla hesaplama birimi (compute unit) bulundurabilir. Günümüzde 2 ila 8 çekirdeğe sahip olan CPU’larda veya 80 ila 100 çerkirdeğe sahip olabilen GPU’larda her bir çekirdek bir hesaplama birimine karşılık düşer.

3.1.2 Çekirdek fonksiyonları

C tabanlı OpenCL dili ile OpenCL destekleyen aygıtlar üzerindeki hesaplama birimleriyle çalıştırılmak üzere yazılan fonksiyonlar çekirdek fonksiyonlarıdır. Çekirdek fonksiyonları C, C++, Objective C gibi dillerde yazılabilen ev sahibi programlardan OpenCL API çağrıları aracılığıyla ile tetiklenir ve hesaplama birimleri üzerinde çalışır, sonuçları yine ev sahibi programa döndürür. Çekirdek fonksiyonlar, çalışma zamanı (runtime) esnasında, hesaplama birimi üzerinde çalışacak şekilde kaynak kodlarından derlenir. Çekirdek fonksiyonu derlendiğinde bir çekirdek oluşur.

3.1.3 Çekirdek nesneleri

Çekirdek nesneleri program içerisinde tanımlanmış belli bir çekirdeği ve o çekirdek ile çalıştırılan argüman değerlerini tutar.

3.1.4 Programlar

Bir OpenCL programı, OpenCL çekirdeklerini, bu çekirdekler tarafından çağırılan dış fonksiyonları ve sabitleri barındırır.

3.1.5 Bağlamlar

Bağlam (context) OpenCL çekirdeklerinin yürütüldüğü ortamı temsil eder. Bağlam, aygıt kümesini, bu aygıtların erişebildiği bellek bilgisini ve çekirdekler üzerinde yürütülmesi için zamanlanmış komut kuyruklarının (command queue) bilgisini tutar. Bağlam, bellek nesnelerinin aygıtlar arasında paylaşılmasını sağlar.

3.1.6 Program nesneleri

Bir program nesnesi, bir OpenCL programını temsil eden veri tipidir. Program bağlamı referansı, program kaynak kodu, programın derlenmiş ve yürütülebilir hali, programın hangi aygıtlar için derlendiği, derlenme seçenekleri ve derlenme kaydı (build log) bilgileri program nesnesinde tutulur.

3.1.7 Komut kuyrukları

Komut kuyrukları hesaplama aygıtlarına iş atamak için kullanılır. Aygıtlardaki çekirdeklerin yürütülmesini ve bellek nesnelerini düzenlerler. OpenCL komutları, komut kuyruğundaki sıraya göre yürütür.

3.1.8 Ev sahibi programlar

Ev sahibi program, çekirdek fonksiyonların hesaplama aygıtı üzerinde yürütülmesi için gerekli bağlamı hazırlayan ve yürütme işlemini düzenleyen programdır. Ev sahibi program CPU üzerinde çalışır fakat CPU aynı zamanda bir hesaplama aygıtı olarak da kullanılabilir. Çekirdek fonksiyonların aygıtlar üzerinde çalışması için ev sahibi program hangi hesaplama aygıtlarının kullanılabileceğini bulur, uygulama için uygun olan hesaplama aygıtını seçer, seçilen aygıtlar için komut kuyruklarını oluşturur ve çekirdek fonksiyonlarda kullanılacak bellek nesnelerini yaratır.

3.1.9 Bellek nesneleri

Bellek nesneleri aygıtın genel belleğinin ilgili bölgelerine referanstır. Ev sahibi program bellek nesnelerini kullanarak aygıt belleğine yazma ve aygıt belleğinden okuma işlemlerini gerçekleştirebilir.

3.2 OpenCL Çalışma Modeli

OpenCL çalışma modeli aşağıdaki modellerin birleşimi şeklinde tanımlanır:

- OpenCL platform modeli
- OpenCL yürütme modeli
- OpenCL bellek modeli
- OpenCL programlama modeli

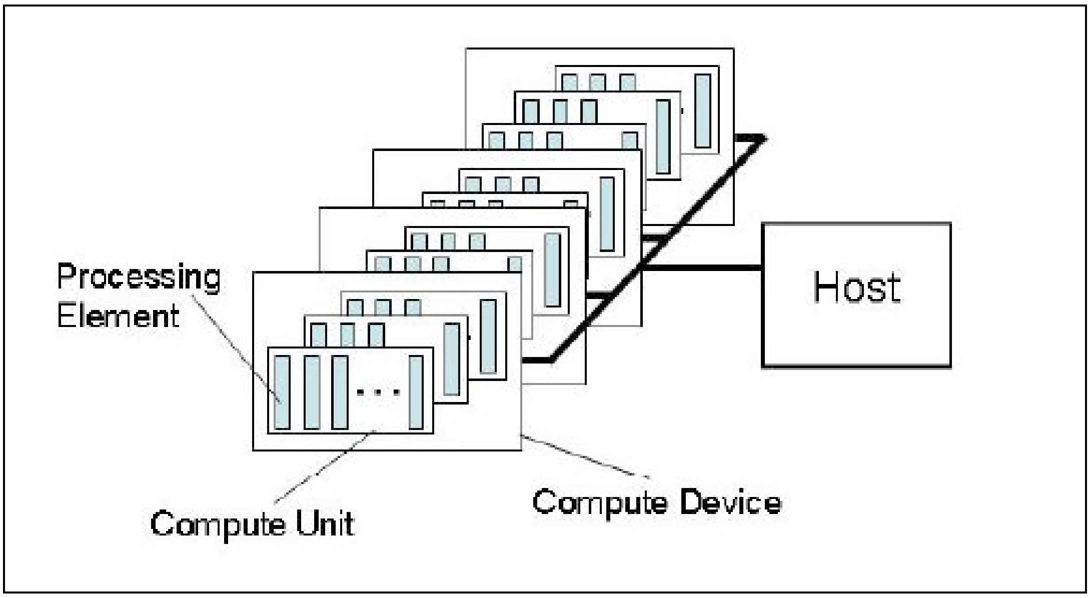
3.2.1 OpenCL platform modeli

OpenCL platform modeli bir ev sahibi program ve onun bağlandığı bir veya daha fazla OpenCL aygıtından oluşur. Bir aygıt bir veya birden fazla hesaplama birimi, her bir hesaplama birimi de bir veya birden fazla işleme elemanı (processing element) barındırır. Aygıt üzerindeki hesaplamalar, işleme elemanlarında gerçekleşir.

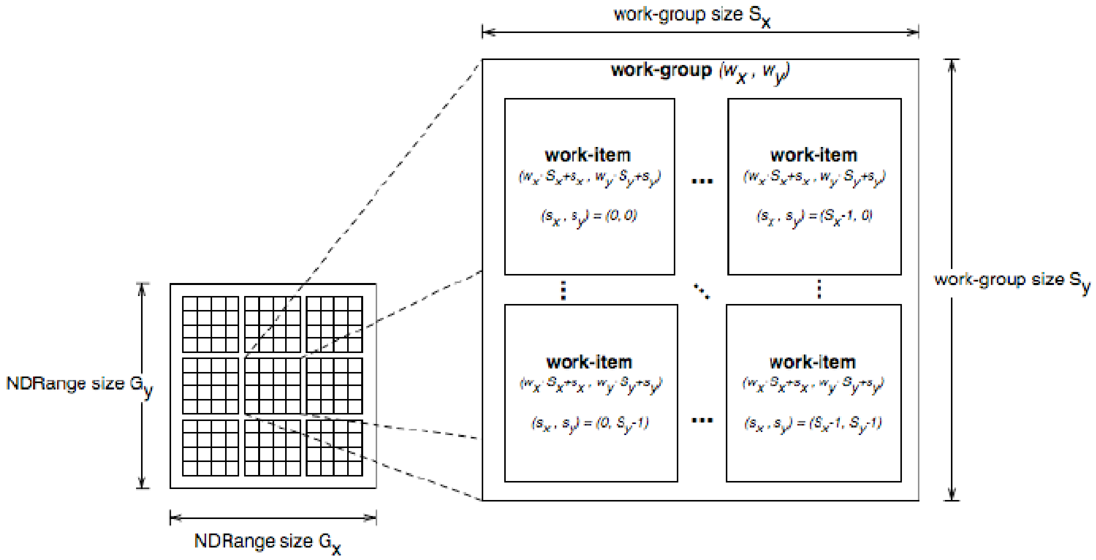
Ev sahibi program komut kuyruklarını kullanarak işleme birimlerine çekirdek fonksiyonların yürütülmesi için komutlar gönderir. Hesaplamalar, ev sahibi programın yarattığı bellek nesneleri üzerinde, işleme elemanlarında SIMD yapısında gerçekleştirilir. Ev sahibi program yine komut kuyruğu aracılıyla hesaplama sonuçlarının bulunduğu aygıt belleğini okuyabilir. Şekil 3.1’de OpenCL Platform Modeli gösterilmiştir.

3.2.2 OpenCL yürütme modeli

OpenCL yürütme modeli bir veya birden fazla hesaplama aygıtı üzerindeki çekirdek örneklerinin (kernel instances) ev sahibi program tarafından eş zamanlı olarak işletilmesini kapsar. Çekirdeğin her bir örneği bir iş öğesi (work-item) olarak tanımlanmaktadır. İş öğeleri GPU çekirdekleri tarafından eş zamanlı olarak, her bir çekirdek bir iş çğesini çalıştıracak şekilde yürütülür. Her bir iş öğesi aynı çekirdek fonksiyonunu kendisine düşen veri parçacığı üzerinde yürütür. İş öğeleri bir araya gelerek iş gruplarını (work group) oluşturur. Aynı zamanda, tüm veriyi kapsayan iş öğeleri bir araya gelerek bir endeks alanı (index space) tanımlar. OpenCL 1, 2 ve 3 boyutlu endeks alanlarını destekler. Bu endeks alanı OpenCL standartlarında NDRange endeks alanı olarak adkandırılır. Endeks alanı içerisindeki her iş öğesi, genel ID'sini (global ID) veya yerel ID (local ID) ve iş gubu ID (work group ID) kullanarak verinin hangi kısmını işleyeceğini belirler. Şekil 3.2 NDRange endeks alanını, iş gruplarını ve iş öğelerini göstermektedir.



Şekil 3.1 : OpenCL platform modeli [15].



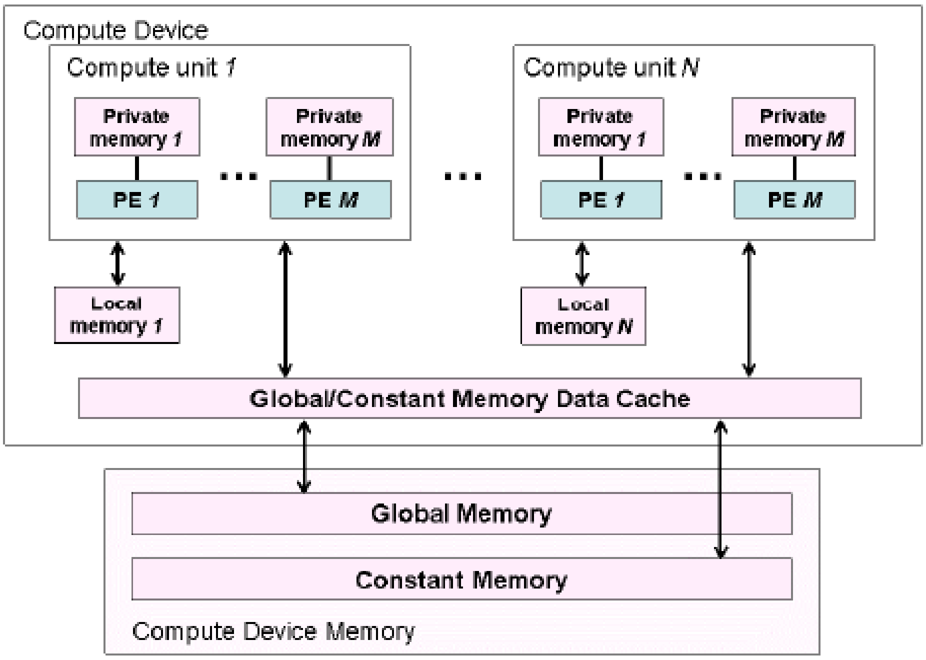
Şekil 3.2 : NDRange endeks alanı, iş grupları ve iş öğeleri [16].

3.2.3 OpenCL bellek modeli

Ev sahibi program bellek nesnelerini OpenCL API çağrılarıyla genel bellek (global memory) üzerinde yaratır. Ev sahibi programın ve hesaplama aygıtlarının bellekleri genellikle birbirinden bağımsız çalışır. İki bellek arası etkileşim gerektiğinde bellek blokları iki bellek arasında transfer edilir veya ev sahibi uygulama aygıt belleğine erişim için haritalama (mapping / unmapping) yöntemini kullanır. OpenCL bellek modelinde, iş öğelerinin erişebileceği dört çeşit bellek alanı vardır:

- Genel bellek (global memory): Tüm iş öğelerinin okuma ve yazma için erişimine açık olan bellek bölgesidir.
- Sabit bellek (constant memory): Genel belleğin, çekirdek fonksiyonlarının yürütülmesi esnasında sabit kalan alanıdır. Ev sahibi programın, çekirdek fonksiyonunun yürütülmesinden önce bu bölgeye yazdığı veri tüm iş öğeleri tarafından okunabilir.
- Yerel bellek (local memory): Bir iş grubu içerisinde paylaşılan, tüm iş öğelerinin okuma ve yazma iznine sahip oldukları bellek bölgesidir.
- Özel bellek (private memory): Bir iş öğesinin kendine özel bellek bölgesidir. Diğer iş öğeleri bu kısma erişemezler.

Şekil 3.3'te OpenCL bellek modeline göre hesaplama aygıtı içerisindeki bellek bölgesi çeşitleri gösterilmiştir.



Şekil 3.3 : Bellek bölgesi çeşitleri [17].

3.2.4 OpenCL programlama modeli

OpenCL, veri palalel (data parallel) ve görev paralel (task parallel) programlama modellerini destekler. Veri paralel programlama modelinde aynı çekirdek fonsiyonu verinin küçük parçaları üzerinde paralel olarak yürütülür. Her veri kümesi 1, 2 veya 3 boyutlu uzayda belirli noktalar kümesine karşılık düşer. Görev paralel programlama modelinde ise farklı çekirdek fonksiyonları yaratılır ve bu fonksiyonlar farklı verilerle aynı anda paralel çalışan iplikler şeklinde yürütüler.

4. JSON-RPC

JSON – RPC, RPC komutlarının sunucu (server) ve istemci (client) arasında JSON (JavaScript Object Notation) formatına göre kodlanmış (JSON - encoded) olarak transfer edilmesini sağlayan bir uzak yordam çağırma (RPC – Remote Procedure Call) protokolüdür. Uzak yordam verisi JSON kodlamaya göre UTF-8 katarlar (string) halinde kodlanır, HTTP veya TCP/IP iletişim protokolleri kullanılarak sunucu ve istemci arasında iletilir.

Bu tez çalışmasında sunucu ve istemci arasındaki veri ve komut iletişimini sağlamak için JSON – RPC protokolü kullanılmıştır. JSON – RPC protokolü verileri metin tabanlı mesajlara çevirip aktardığından işletim sistemi bağımsız olarak çalışabilmektedir. JSON – RPC protokolünü Microsoft RPC, Java RMI, XML – RPC gibi benzer teknolojilerden farklı kılan işletim sistemi bağımsız olarak makul bir hızda çalışabilmesi, gerçeklemesinin kolay olması ve oluşan mesajların transfer boyutu açısından uygun olmasıdır.

Tez çalışmasında JSON – RPC protokolünü gerçeklemek için C++ dilinde yazılmış olan açık kaynaklı JSONRPC – CPP çatısı kullanılmışdır [18]. İletişimi hızlandırmak ve ölçeklendirmek amacıyla çatı üzerinde bazı değişiklikler yapılmıştır. Ayrıca veri tiplerinin düzgün ve minimum yer kaplayacak şekilde transfer edilebilmesini sağlamak için bazı tip dönüşümü ve kodlama yöntemleri uygulanmıştır. Bu değişiklikler ve yöntemler şu şekildedir:

- Mesaj boyutunu düşürmek için *host_id, method, jsonrpc* (versiyon) gibi zorunlu parametreler kaldırılmış veya en az boyut kaplayacak hale kısaltılmıştır. Bununla birlikte her fonksiyon için parametre adları ve diğer değerler mümkün olan en kısa şekilde yazılmıştır.
- Boyutu maksimum izin verilen TCP paket boyutundan daha büyük olan mesajların istemci tarafında bölünüp, sunucu tarafında doğru bir şekilde tekrar birleştirilip işlenmesi gerçekleştirilerek bu mesajların da iletilebilmesi sağlanmıştır.

- OpenCL tipleri JSON mesajı ile aktarılabilecek C tiplerine dönüştürüp, mesajı alan tarafta tekrar OpenCL tiplerine geri dönüştürülmüştür. Çizelge 4.1 tip dönüşümlerini göstermektedir.
- *Byte* dizileri şeklinde olan parametreler mesaj boyutunu azaltmak ve katar (string) şeklinde taşınmak için *base64* kodlaması ile kodlanmıştır.
- Kodlanmış *byte* dizilerinin boyutunu azaltmak için çeşitli katar sıkıştırma algoritmaları denenmiş fakat sıkıştırmak için harcanan zamanın transfer zamanından sağlanan kazanca göre daha fazla olması nedeniyle bu yöntemler uygulanmamıştır.
- TCP bağlantı açma ve kapatma maliyetlerini en aza indirmek amacıyla sunucu ve istemci arasındaki TCP bağlantısı ilk çağrıda açılmış ve ev sahibi program sonandığında kapatılmıştır.

Çizelge 4.1 : OpenCL tiplerinin ve C tiplerine dönüşümü

OpenCL Tipi	Karşılık Düşen C++ Tipi	Kodlama Tekniği
Tam sayı tipleri ör: cl_int, cl_uint, cl_bool, cl_ulong	int, unsigned int, long unsigned int	-
İşaretçi tipleri ör: cl_platform_id, cl_device_id, cl_context, cl_mem, cl_program, cl_kernel	İşaretçi adresleri unsigned long int tipine dönüştürüldü	-
Veri parametreleri ör: float array, integer array, char array	İşaretçilerinden ve uzunluklarından yararlanılarak karakter dizisine dönüştürüldü	Base64 kodlaması ile kodlandı

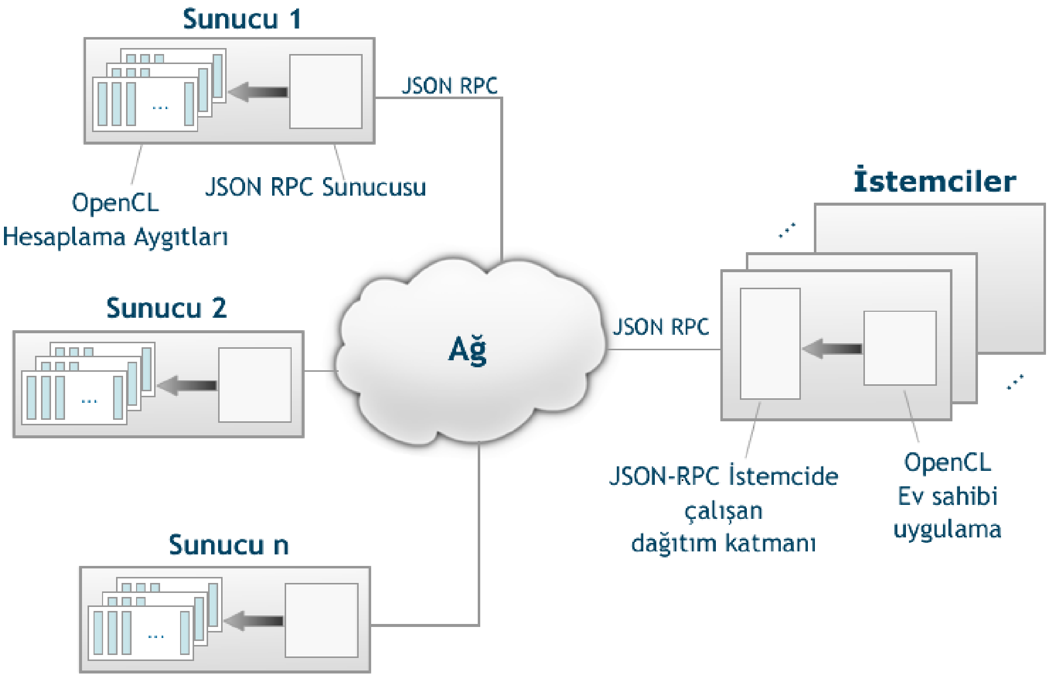
5. DISTRIBUTED OPENCL

5.1 Distributed OpenCL Genel Özellikleri

Distributed OpenCL (Dağıtılmış OpenCL); bu tez çalışması kapsamında geliştirilmiş olan, istemci – sunucu mimarisinde, birden çok istemci veya birden sunucudan oluşabilen, OpenCL avantajlarını ve işlevselliğini koruyarak OpenCL'i ağ ölçeğine genişletip birden çok OpenCL çalıştırabilen bilgisayarın aynı problem üzerinde paralel olarak çalışabilmesini sağlayan bir çatıdır. Distributed OpenCL çatısının amacı aynı problem üzerinde çalışan hesaplama aygıtı sayısını arttırarak genel işlem süresini azaltmaktır. Distributed OpenCL çatısı aşağıdaki bileşenlerden oluşmaktadır:

- İstemciler
- Dağıtım Katmanı
- Sunucular

Şekil 5.1'de Distributed OpenCL çatısının mimarisi gösterilmektedir.



Şekil 5.1 : Distributed OpenCL çatısı mimarisi.

İstemciler OpenCL ev sahibi uygulamalarını (OpenCL host application) çalıştırır ve ev sahibi uygulamadan yapılan OpenCL API çağrıları istemci üzerinde çalışan dağıtım katmanı (distribution layer) aracılığıyla ağ üzerindeki sunuculara yönlendirilir. OpenCL ile ilgili işlemler ise sunucularda gerçekleştirilir.

İstemci üzerinde çalışan ev sahibi uygulama bir OpenCL API çağrısı yaptığında bu çağrı yine istemci üzerinde çalışan dağıtım katmanına iletilir. Dağıtım katmanı API çağrılarını JSON mesajlarına dönüştürür ve bu mesajlar JSON RPC iletişim protokolü kullanılarak sistemdeki ilgili sunuculara aktarılır. Sunucuda çalışan JSON – RPC sunucu uygulaması kendisine gelen mesajları dinler ve bu mesajları sunucu üzerindeki OpenCL aygıtlarıyla çalışacak OpenCL çağrılarına dönüştürür. Bu yöntemle OpenCL API çağrıları istemcideki parametrelerle sunucuda gerçeklenmiş olur. Sunucu uygulaması OpenCL hesaplama aygıtından gelen cevapları JSON mesajına dönüştürerek API çağrısını yapan istemciye yollar. OpenCL aygıtları, OpenCL fonksiyonlarında kullanılan bağlamı ve durum bilgisini kendi belleklerinde tuttuklarından sunucuda OpenCL fonksiyonlarına ait herhangi bir bağlam veya durum bilgisi tutulmaz. *cl_mem, cl_context, cl_program, cl_kernel* gibi gerçeklenmesi üreticinin OpenCL kütüphanesine bağlı olan ve OpenCL çağrıları tarafından yaratılan OpenCL tiplerinin örneklerine (instance) ait işaretçiler sunucu uygulamada belleğin *heap* kısmında yaratılırlar. Bu şekilde fonksiyon kapsamı dışında (function scope) da yok edilmedikleri ve erişilebilir oldukları için daha sonraki OpenCL API çağrılarında da işaretçi adresleri kullanılarak ilgili örneğe erişilebilir. Bu yaratılan bellek alanları ev sahibi tarafından *clReleaseMemObject, clReleaseContext* gibi çağrılar yapıldığında bu çağrıları sunucu tarafında gerçekleyen fonksiyonlar tarafından işletim sistemine geri verilir. Bu bellek alanlarına ait işaretçiler sunucu uygulamada çağrıyı yapan ev sahibi programın numarası (host ID) ile eşleştirilerek işaretçi haritalarında (pointer map) tutulurlar.

GPU mimarisinde işlemlerin SIMD modeli çerçevesinde işlendiği düşünüldüğünde işlenecek olan verinin sistemdeki çok sayıda sunucuya dağıtılabilmesi olanaklıdır. Bu özellik sunucu başına düşen iş yükünü dolayısıyla da işleme elemanı başına düşen iş yükünü azaltıp paralellik seviyesini arttırarak hesaplama hızında artışı sağlamaktadır.

5.2 İstemciler

Distributed OpenCL çatısında istemci, istemci makinede çalışan bir veya paralel olarak yürütülen birden fazla ev sahibi uygulamadır (host application). Doğal OpenCL (Native OpenCL) için yazılmış olan ev sahibi uygulamaları Distributed OpenCL çatısı ile çalıştırmak için yapılması gerekenler şu şekildedir:

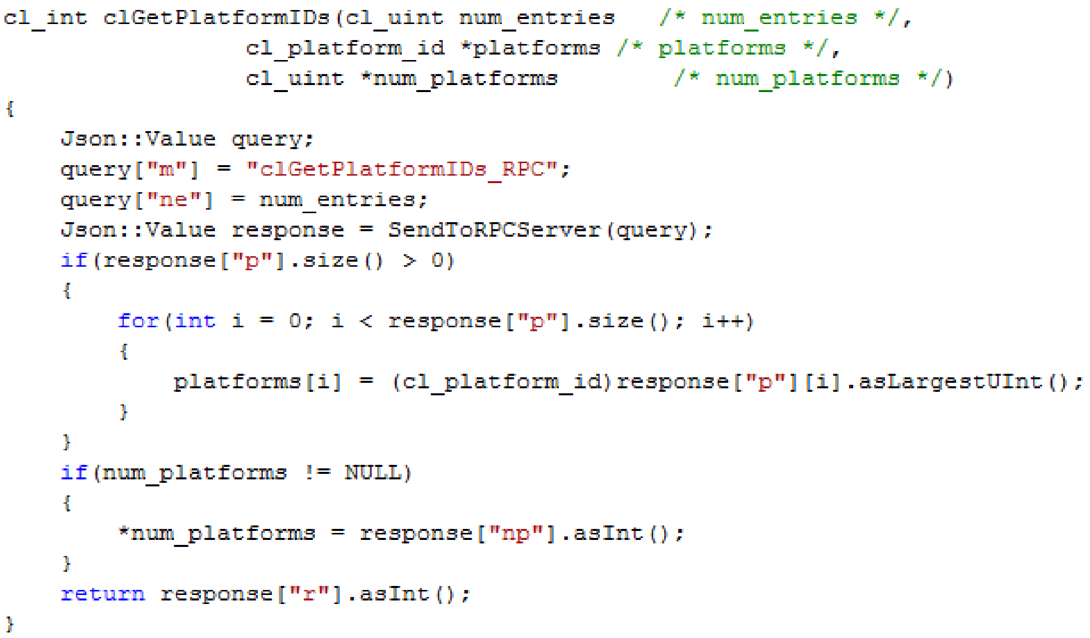
- Ev sahibi programı derlerken OpenCL tiplerinin ve fonksiyon imzalarının bulunduğu OpenCL *include* klasörü yerine Distributed OpenCL dağıtım katmanının *include* klasörü referans olarak gösterilmelidir.
- Bağlama (link) aşamasında OpenCL kütüphanesi yerine Distributed OpenCL dağıtım katmanı kütüphanesi referans gösterilmelidir.

OpenCL ev sahibi uygulama bu şekilde derlenip çalıştırıldığında OpenCL çağrıları üreticinin OpenCL kütüphanesi yerine dağıtım katmanı tarafından yakalanıp işlenmeye başlar.

5.3 Dağıtım Katmanı

Dağıtım katmanı istemci tarafında çalışan Distributed OpenCL kütüphanesidir. Dağıtım katmanı içerisinde OpenCL tiplerini ve fonksiyon imzalarını barındıran orijinal OpenCL başlık (header) dosyalarının kopyaları, bu dosyalardaki fonksiyonları gerçekleyen C++ kaynak dosyaları ile tip dönüşümü, dağıtım ve iletişim ile ilgili işlemlerim gerçekleştirildiği fonksiyonlar bulunur. Başlık dosyaları ve OpenCL çağrılarının gerçeklendiği kısımlar OpenCL v1.1 belirtimi (OpenCL v1.1 specification) göz önünde bulundurularak hazırlanmıştır.

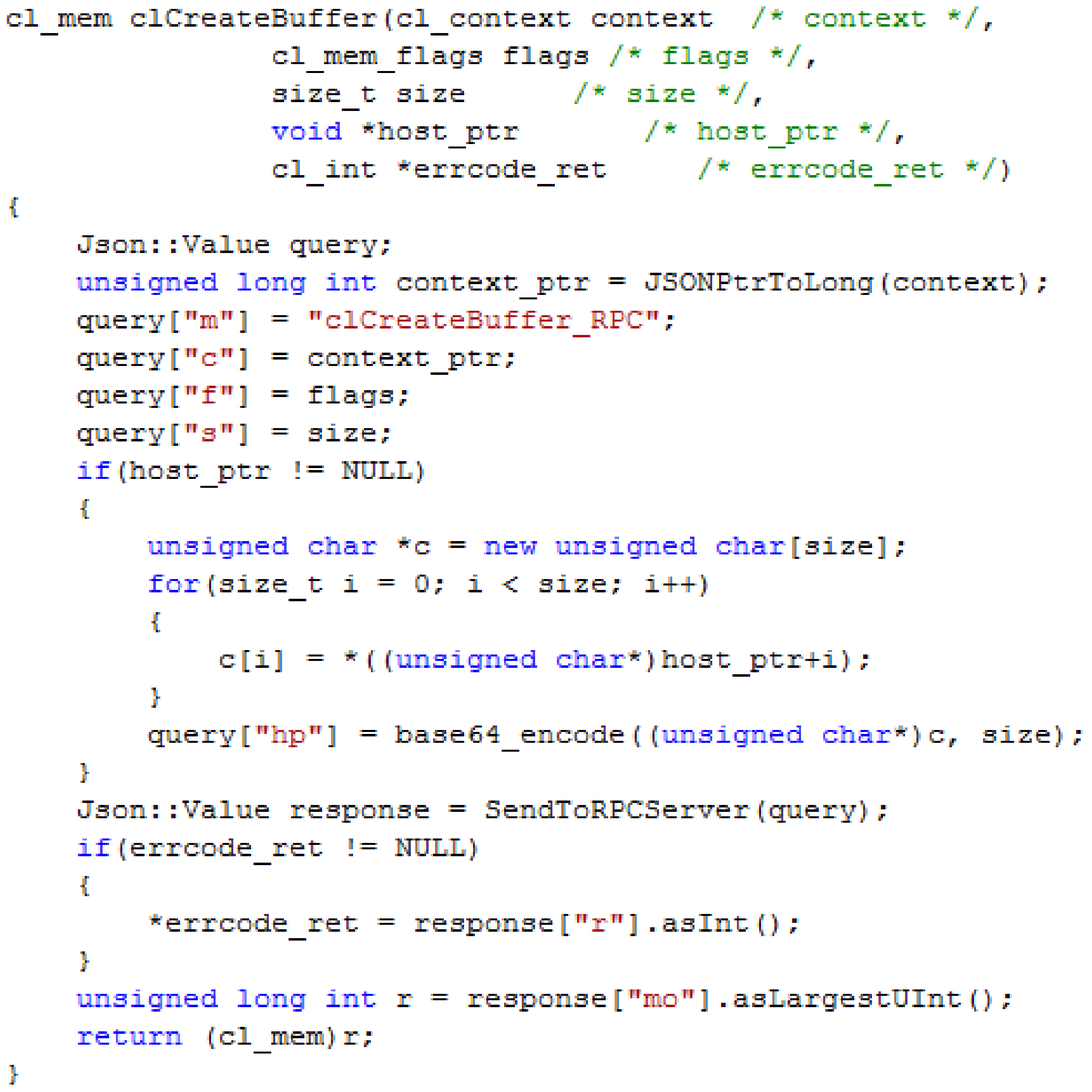
Distributed OpenCL çatısında ev sahibi programın yaptığı OpenCL API çağrıları üreticinin gerçeklediği OpenCL kütüphanesi yerine dağıtım katmanı tarafından yakalanır. Dağıtım katmanı, her bir OpenCL çağrısı için kendisindeki ilgili fonksiyonu yürütür. Bu fonksiyonlar OpenCL tiplerindeki parametrelere çağırılacak fonksiyon ismini ve ev sahibi program numarasını (host id) ekleyerek sunucuya gönderilecek JSON mesajının içeriğini oluştururlar. Bu mesajlar dağıtım katmanında çalışan TCP / IP tabanlı iletişim kuran JSON – RPC istemcisi tarafından sunucuya gönderilir. Daha sonra sunucudan cevap olarak gönen JSON mesajı OpenCL fonksiyonunda kullanılan parametre tiplerine dönüştürülür ve ev sahibi programa döndürülür. İstemci tarafındaki tip dönüşümü işlemleri, veri kodlama ve kod çözme işlemleri de dağıtım katmanı tarafından gerçekleştirilir. Şekil 5.2'de dağıtım katmanında OpenCL fonksiyonlarının gerçeklenmesine örnek olarak *clGetPlatformIDs* fonksiyonunun gerçeklenmesi gösterilmiştir. Fonksiyona ait giriş, dönüş parametrelerinin atanmaları ve kullanımları OpenCL v1.1 belirtiminde açıklanan fonksiyonun yerine getirdiği işleve uygun olarak gerçekleştirilmiştir. Fonksiyonun imzası orijinal OpenCL'deki *clGetPlatformID* fonksiyonu imzasıyla tamamen aynıdır.



```
cl_int clGetPlatformIDs(cl_uint num_entries   /* num_entries */,
                  cl_platform_id *platforms /* platforms */,
                  cl_uint *num_platforms        /* num_platforms */)
{
    Json::Value query;
    query["m"] = "clGetPlatformIDs_RPC";
    query["ne"] = num_entries;
    Json::Value response = SendToRPCServer(query);
    if(response["p"].size() > 0)
    {
        for(int i = 0; i < response["p"].size(); i++)
        {
            platforms[i] = (cl_platform_id)response["p"][i].asLargestUInt();
        }
    }
    if(num_platforms != NULL)
    {
        *num_platforms = response["np"].asInt();
    }
    return response["r"].asInt();
}
```

Şekil 5.2 : Dağıtım katmanında *clGetPlatformIDs* fonksiyonu.

clGetPlatformIDs fonksiyonu *num_entries* parametresini giriş olarak alır, aygıtta bulunan platformların *platform_id* lerine işaretçileri ve aygıttaki platform sayısını (*num_platform*) döndürür. Bu fonksiyonda da sunucu tarafındaki fonksiyon adı olan *clGetPlatformIDs_RPC* ve *num_entries* parametreleri JSON mesajına eklenerek *SendToRPCServer* fonksiyonu çağrılarak sunucuya gönderilmiştir. Daha sonra sunucudan gelen cevap mesajındaki *platform_id* lere işaretçi adreslerini taşıyan *p* anahtarındaki (*reponse[“p”]*) değerler fonksiyon parametresi olan *platforms* işaretçisine (dizisi) aktarılmıştır. *np* anahtarındaki platform sayısı değeri ise *num_platforms* işaretçisine değer olarak atanmıştır. *r* anahtarındaki değer ise *clGetPlatformIDs* fonksiyonunun dönüş değeri olan hata numarasıdır. Şekil 5.3'te ise dağıtım katmanında *clCreateBuffer* fonksiyonun gerçeklenmesi gösterilmiştir. Burada daha önceki örnekten farklı olarak işaretçi tipindeki parametrelerin ve *void* işaretçi tipindeki parametrelerin aktarılması ve işaretçi tipindeki parametrelerin döndürülmesi gösterilmiştir.

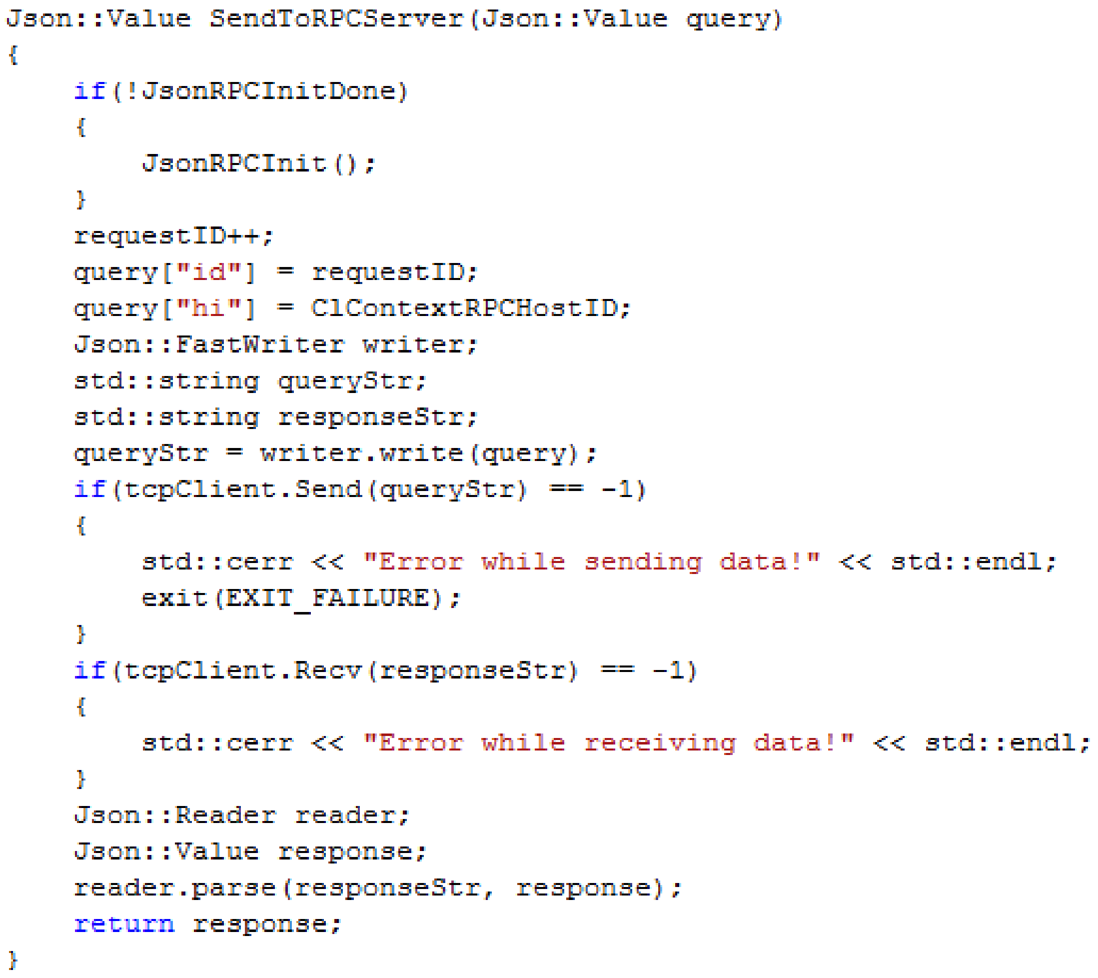


```
cl_mem clCreateBuffer(cl_context context  /* context */,
              cl_mem_flags flags /* flags */,
              size_t size      /* size */,
              void *host_ptr       /* host_ptr */,
              cl_int *errcode_ret     /* errcode_ret */)
{
    Json::Value query;
    unsigned long int context_ptr = JSONPtrToLong(context);
    query["m"] = "clCreateBuffer_RPC";
    query["c"] = context_ptr;
    query["f"] = flags;
    query["s"] = size;
    if(host_ptr != NULL)
    {
        unsigned char *c = new unsigned char[size];
        for(size_t i = 0; i < size; i++)
        {
            c[i] = *((unsigned char*)host_ptr+i);
        }
        query["hp"] = base64_encode((unsigned char*)c, size);
    }
    Json::Value response = SendToRPCServer(query);
    if(errcode_ret != NULL)
    {
        *errcode_ret = response["r"].asInt();
    }
    unsigned long int r = response["mo"].asLargestUInt();
    return (cl_mem)r;
}
```

Şekil 5.3 : Dağıtım katmanında *clCreateBuffer* fonksiyonu.

İşaretçi tipinde olan *context* parametresi unsigned long int tipine dönüştürülerek adres değeri sunucuya gönderilecek mesaja eklenmektedir. *byte* cinsinden büyüklüğü *size* parametresinde gönderilen *host_ptr* parametresi *size* büyüklüğünde bir *unsigned char* dizisine aktarılmakta bu karakter dizisi *base64* olarak kodlanmakta ve mesaja eklenmektedir. Dönüş değeri olan sunucuda çalışan OpenCL API tarafından yaratılan *cl_mem* örneğinin adresi fonksiyonun dönüş değeri olarak ev sahibi programa iletilir.

Şekil 5.4'te dağıtım katmanında OpenCL çağrılarını yakalayan fonksiyonlar tarafından oluşturulan JSON mesajlarını sunucuya gönderen ve sunucudan gelen sonucu alan *SendToRPCServer* fonksiyonu gösterilmiştir. Bu fonksiyonda TCP bağlantısı henüz kurulmamışsa bağlantı kurulur, her gönderme işleminde global *requestID* değişkeni 1 arttırılır, her isteğe bir numara verilir ve bu numara mesaja eklenir. Böylece sunucu tarafında isteğin kimliği anlaşılmış olur. JSON mesajına ev sahibi programın numarası da (*ClContextRPCHostID*) eklendikten sonra mesaj sunucuya gönderilir. Sunucudan gelen cevap da *JSON::Value* tipine dönüştürülerek çağıran fonksiyona döndürülür.



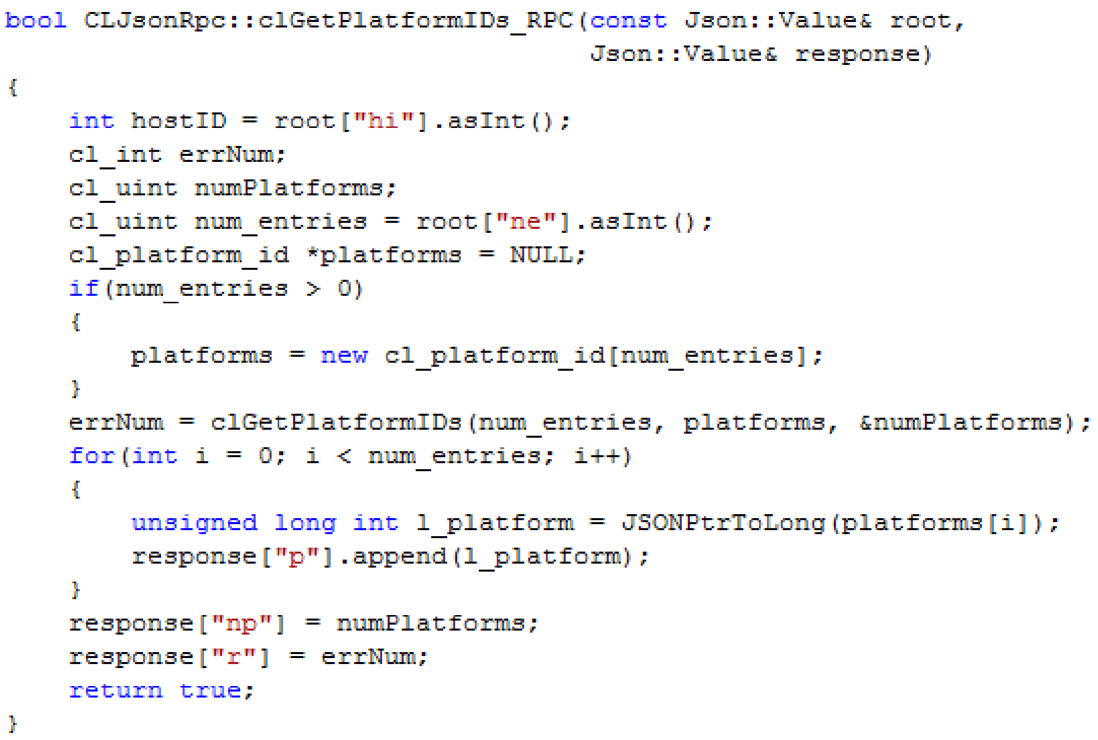
```
Json::Value SendToRPCServer(Json::Value query)
{
    if(!JsonRPCInitDone)
    {
        JsonRPCInit();
    }
    requestID++;
    query["id"] = requestID;
    query["hi"] = ClContextRPCHostID;
    Json::FastWriter writer;
    std::string queryStr;
    std::string responseStr;
    queryStr = writer.write(query);
    if(tcpClient.Send(queryStr) == -1)
    {
        std::cerr << "Error while sending data!" << std::endl;
        exit(EXIT_FAILURE);
    }
    if(tcpClient.Recv(responseStr) == -1)
    {
        std::cerr << "Error while receiving data!" << std::endl;
    }
    Json::Reader reader;
    Json::Value response;
    reader.parse(responseStr, response);
    return response;
}
```

Şekil 5.4 : Dağıtım katmanında JSON mesajının sunucuya gönderilmesi.

5.4 Sunucular

Sunucularda çalışan JSON – RPC sunucu uygulaması gelen JSON mesajlarını yakalamak için ilgili portları dinler. Bir mesaj geldiğinde eğer mesajın devamı varsa beklenir ve devam mesajları birleştirilir. Sonraki aşamada mesajın içeriğindeki çağrılan fonksiyon ismine bakılır ve ona göre ilgili fonksiyon çağrılır. JSON mesajındaki parametreler üzeride tip dönüşümü ve varsa kod çözme (*decoding*) işlemleri gerçekleştirildikten sonra bu parametrelerle orijinal OpenCL API çağrısı yapılır. Sunucu üzerinde bulunan OpenCL aygıtları hesaplamaları yaptıktan sonra sonuç değerleri yine JSON mesajı haline getirilerek istemciye cevap yollanır. Sunucuyla istemci arasındaki TCP bağlantısı ev sahibi programın yürütülmesi boyunca açık kalır. Sistemdeki her bir sunucu eş zamanlı olarak birden fazla istemciye veya birden fazla ev sahibi programa hizmet edebilmektedirler.

Şekil 5.5'te *clGetPlatformIDs* fonksiyonunun sunucu tarafında gerçeklenmesi gösterilmiştir.

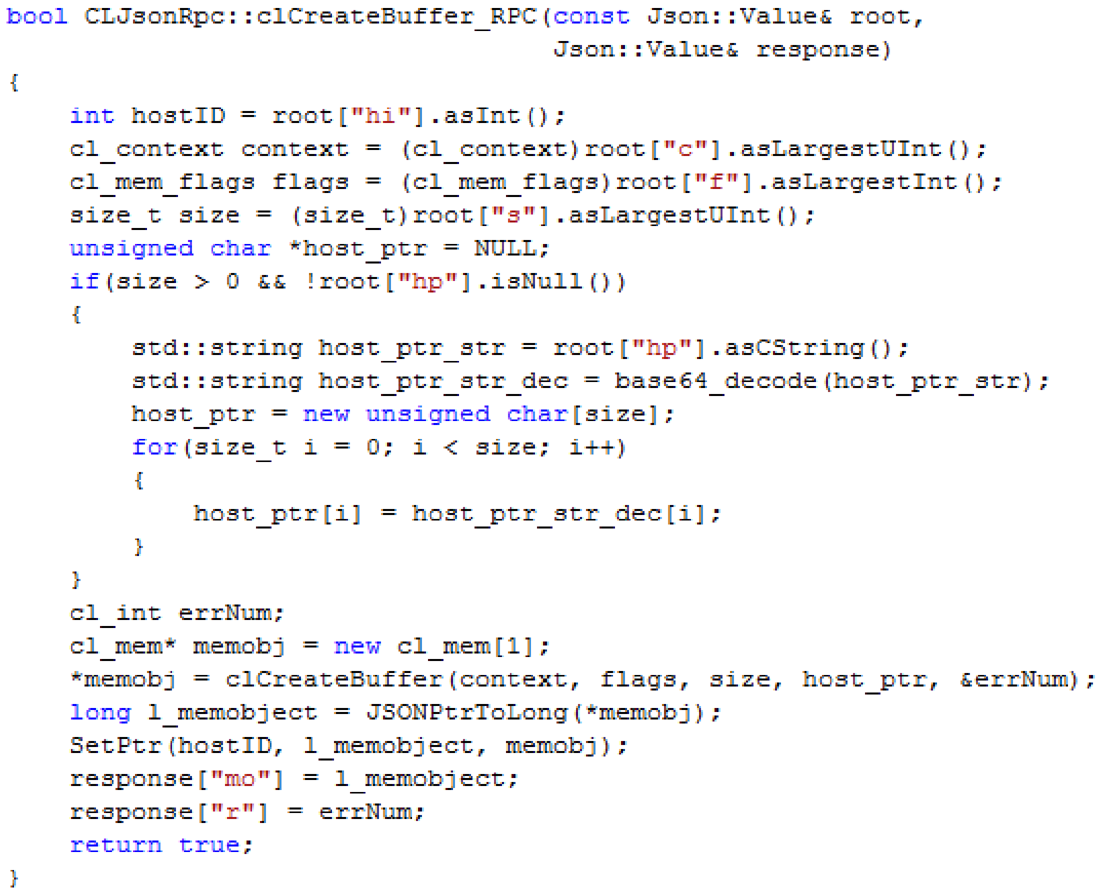


```
bool CLJsonRpc::clGetPlatformIDs_RPC(const Json::Value& root,
                                     Json::Value& response)
{
    int hostID = root["hi"].asInt();
    cl_int errNum;
    cl_uint numPlatforms;
    cl_uint num_entries = root["ne"].asInt();
    cl_platform_id *platforms = NULL;
    if(num_entries > 0)
    {
        platforms = new cl_platform_id[num_entries];
    }
    errNum = clGetPlatformIDs(num_entries, platforms, &numPlatforms);
    for(int i = 0; i < num_entries; i++)
    {
        unsigned long int l_platform = JSONPtrToLong(platforms[i]);
        response["p"].append(l_platform);
    }
    response["np"] = numPlatforms;
    response["r"] = errNum;
    return true;
}
```

Şekil 5.5 : Sunucuda *clGetPlatformIDs* fonksiyonu.

Fonksiyonun giriş olarak aldığı *num_entries* parametresi JSON mesajından alınmıştır. *platforms* parametresi için de *num_entries* parametresindeki değere göre bellek alanı yaratılmıştır. OpenCL API çağrısı ile *clGetPlatformIDs* fonksiyonu çağrılmış, dönen varsa platform listesindeki elemanlar, platform sayısını belirten *numPlatforms* değeri ve fonksiyonun dönüşü olan hata numarası değeri cevap mesajına eklenmiştir. Bu aşamadan sonra dönüş mesajı sunucuda çalışan JSON – RPC sunucu tarafından isteği gönderen istemciye iletilir.

Şekil 5.6'da sunucuda gerçeklenen *clCreateBuffer* fonksiyonu gösterilmektedir. Daha önceki örnekten farklı olarak bu fonksiyonda katar olarak gelen *hp* anahtarındaki değer *base64* kod çözme işleminden geçirilerek OpenCL fonksiyonuna parametre olarak geçilecek olan *host_ptr* dizisi oluşturuluyor. İstemci tarafında *host_ptr* işaretçisi üzerinden atanan dizi elemanlarının tipi ne olursa olsun, OpenCL fonksiyonu dizi uzunluğunu *byte* cinsinden beklediği için burada *unsigned char* dizisi yaratılması işlevselliği hiç bir şekilde etkilemiyor. *clCreateBuffer* işlemi sonucunda yaratılan *cl_mem* tipindeki örneğe işaret eden işaretçi daha sonraki fonksiyonlarda kullanılmak üzere *host_id* ile eşleştirilmiş olan işaretçi haritasına (*pointer map*) işaretçinin adresi anahtar olacak şekilde ekleniyor.

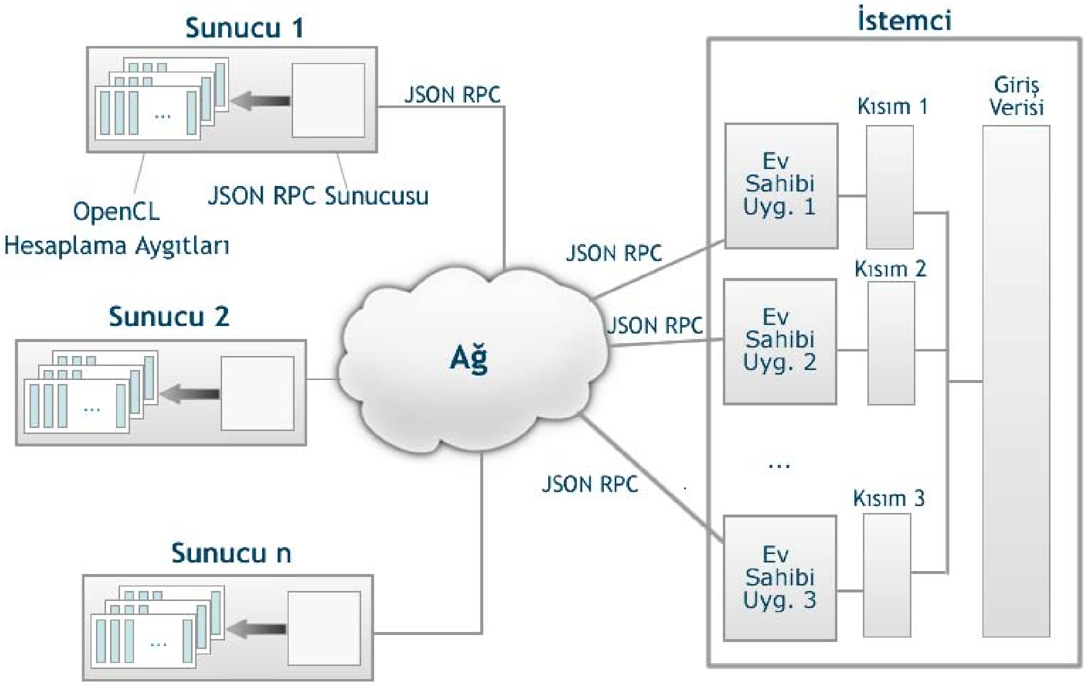


```
bool CLJsonRpc::clCreateBuffer_RPC(const Json::Value& root,
                                   Json::Value& response)
{
    int hostID = root["hi"].asInt();
    cl_context context = (cl_context)root["c"].asLargestUInt();
    cl_mem_flags flags = (cl_mem_flags)root["f"].asLargestInt();
    size_t size = (size_t)root["s"].asLargestUInt();
    unsigned char *host_ptr = NULL;
    if(size > 0 && !root["hp"].isNull())
    {
        std::string host_ptr_str = root["hp"].asCString();
        std::string host_ptr_str_dec = base64_decode(host_ptr_str);
        host_ptr = new unsigned char[size];
        for(size_t i = 0; i < size; i++)
        {
            host_ptr[i] = host_ptr_str_dec[i];
        }
    }
    cl_int errNum;
    cl_mem* memobj = new cl_mem[1];
    *memobj = clCreateBuffer(context, flags, size, host_ptr, &errNum);
    long l_memobject = JSONPtrToLong(*memobj);
    SetPtr(hostID, l_memobject, memobj);
    response["mo"] = l_memobject;
    response["r"] = errNum;
    return true;
}
```

Şekil 5.6 : Sunucuda *clCreateBuffer* fonksiyonu.

5.5 Paralel Yürütmenin Sağlanması

GPU'lar işlemleri SIMD modelinde yürüttüğünden çoğu OpenCL uygulaması sadece giriş verisini daha küçük bloklara ayırarak paralelleştirilebilir. Bunun dışındaki kenar tesbit etme (edge detection), konvolusyon (convolution) gibi işlemlerin bir çoğu da veriyi daha küçük fakat örtüşen bloklara ayırarak paralelleştirilebilir. Distributed OpenCL çatısı, ev sahibi uygulamaların paralel olarak çalışacak şekle dönüştürülmesiyle birlikte ağ üzerindeki birden fazla hesaplama aygıtı üzerinde paralel işlem yürütmeyi sağlamaktadır. Şekil 5.7 Distributed OpenCL çatısında paralel yürütmeyi mimarisini göstermektedir.



Şekil 5.7 : Paralel yürütme mimarisi.

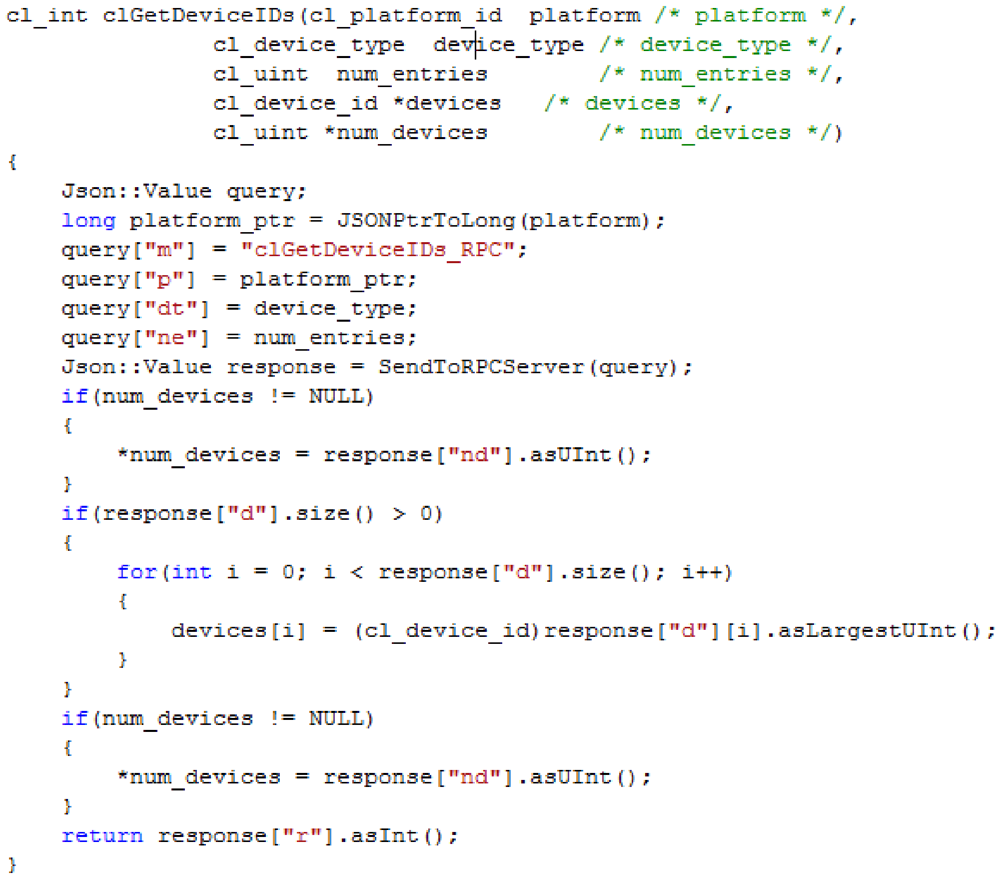
Giriş verisi paralel çalıştırılacak sunucu sayısı kadar parçaya ayrılarak her bir parça için ev sahibi uygulamalar çok iplikli bir yapıda çalıştırılır. OpenCL'e gönderilen ve sonuç olarak dönen verinin yazılması ve okunması için paylaşımlı bellek veya benzer yapılar kullanılabilir.

5.6 Başlıca OpenCL Fonksiyonlarının Gerçeklenmesi

Bu bölümde OpenCL v1.1 belirtiminde yer alan başlıca OpenCL API fonksiyonlarının Distributed OpenCL çatısında sunucu ve istemci tarafında gerçeklenişi gösterilmektedir.

Şekil 5.8 *clGetDevicesIDs* fonksiyonunun dağıtım katmanında istemci tarafında gerçeklenmesini göstermektedir. Platform işaretçisi, aygıt tipi ve giriş sayısı parametreleri sunucuya gönderilip cevap olarak dönen aygıt işaretçileri ve aygıt sayısı sonuçları ev sahibi programa yollanmaktadır.

Şekil 5.9 *clGetDevicesIDs* fonksiyonunun dağıtım katmanında sunucu tarafında gerçeklenmesini göstermektedir. İstemci tarafından gönderilen platform işaretçisi, aygıt tipi ve giriş sayısı parametreleri ile aynı isimli OpenCL fonksiyonu çağrısı yürütülüp dönüş parametreleri istemciye yollanmaktadır.



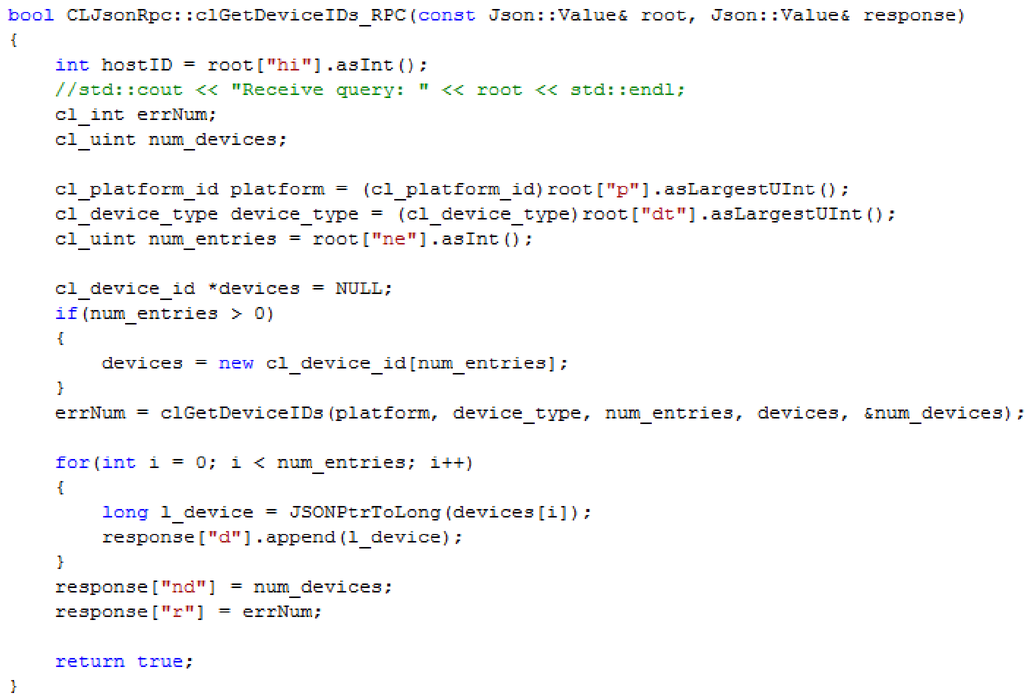
```
cl_int clGetDeviceIDs(cl_platform_id  platform /* platform */,
                      cl_device_type  device_type /* device_type */,
                      cl_uint   num_entries        /* num_entries */,
                      cl_device_id *devices    /* devices */,
                      cl_uint *num_devices         /* num_devices */)
{
    Json::Value query;
    long platform_ptr = JSONPtrToLong(platform);
    query["m"] = "clGetDeviceIDs_RPC";
    query["p"] = platform_ptr;
    query["dt"] = device_type;
    query["ne"] = num_entries;
    Json::Value response = SendToRPCServer(query);
    if(num_devices != NULL)
    {
        *num_devices = response["nd"].asUInt();
    }
    if(response["d"].size() > 0)
    {
        for(int i = 0; i < response["d"].size(); i++)
        {
            devices[i] = (cl_device_id)response["d"][i].asLargestUInt();
        }
    }
    if(num_devices != NULL)
    {
        *num_devices = response["nd"].asUInt();
    }
    return response["r"].asInt();
}
```

Şekil 5.8 : İstemci tarafında *clGetDeviceIDs* fonksiyonu.

Şekil 5.10 *clCreateCommandQueue* fonksiyonunun dağıtım katmanında istemci tarafında gerçeklenmesini göstermektedir. Bağlam işaretçisi, aygıt işaretçisi, komut kuyruğu özellikleri parametreleri sunucuya gönderilip cevap olarak dönen komut kuyruğu işaretçisi ev sahibi programa yollanmaktadır.

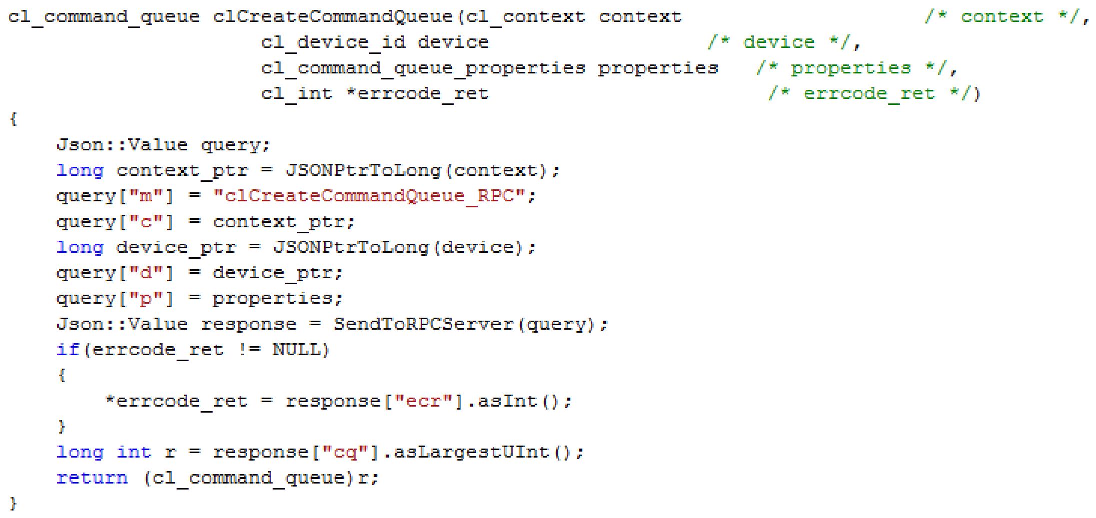
Şekil 5.11 *clCreateCommandQueue* fonksiyonunun dağıtım katmanında sunucu tarafında gerçeklenmesini göstermektedir. İstemci tarafından gönderilen bağlam işaretçisi, aygıt işaretçisi, komut kuyruğu özellikleri parametreleri ile aynı isimli OpenCL fonksiyonu çağrısı yürütülüp dönüş parametreleri istemciye yollanmaktadır.

Şekil 5.12 *clCreateKernel* fonksiyonunun dağıtım katmanında istemci tarafında gerçeklenmesini göstermektedir. Program işaretçisi ve çekirdek adı parametreleri sunucuya gönderilip cevap olarak dönen çekirdek işaretçisi ev sahibi programa yollanmaktadır.



```
bool CLJsonRpc::clGetDeviceIDs_RPC(const Json::Value& root, Json::Value& response)
{
    int hostID = root["hi"].asInt();
    //std::cout << "Receive query: " << root << std::endl;
    cl_int errNum;
    cl_uint num_devices;

    cl_platform_id platform = (cl_platform_id)root["p"].asLargestUInt();
    cl_device_type device_type = (cl_device_type)root["dt"].asLargestUInt();
    cl_uint num_entries = root["ne"].asInt();

    cl_device_id *devices = NULL;
    if(num_entries > 0)
    {
        devices = new cl_device_id[num_entries];
    }
    errNum = clGetDeviceIDs(platform, device_type, num_entries, devices, &num_devices);

    for(int i = 0; i < num_entries; i++)
    {
        long l_device = JSONPtrToLong(devices[i]);
        response["d"].append(l_device);
    }
    response["nd"] = num_devices;
    response["r"] = errNum;

    return true;
}
```

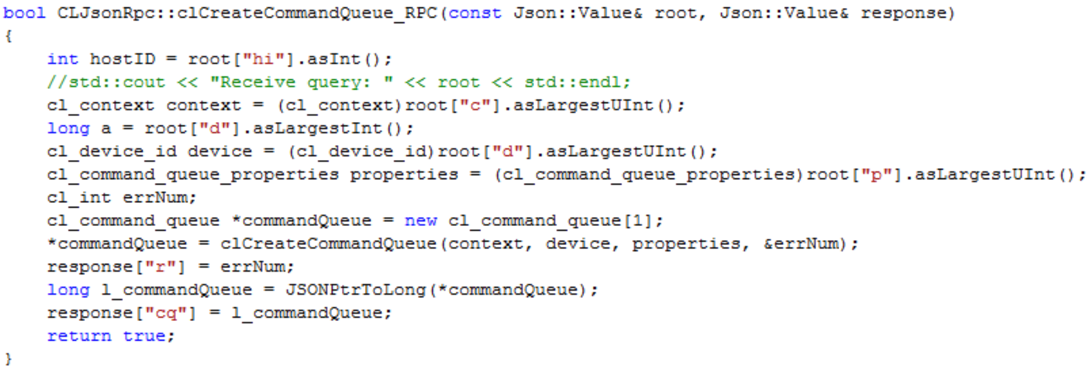
Şekil 5.9 : Sunucu tarafında *clGetDeviceIDs* fonksiyonu.



```
cl_command_queue clCreateCommandQueue(cl_context context                    /* context */,
                     cl_device_id device                  /* device */,
                     cl_command_queue_properties properties   /* properties */,
                     cl_int *errcode_ret                       /* errcode_ret */)
{
    Json::Value query;
    long context_ptr = JSONPtrToLong(context);
    query["m"] = "clCreateCommandQueue_RPC";
    query["c"] = context_ptr;
    long device_ptr = JSONPtrToLong(device);
    query["d"] = device_ptr;
    query["p"] = properties;
    Json::Value response = SendToRPCServer(query);
    if(errcode_ret != NULL)
    {
        *errcode_ret = response["ecr"].asInt();
    }
    long int r = response["cq"].asLargestUInt();
    return (cl_command_queue)r;
}
```

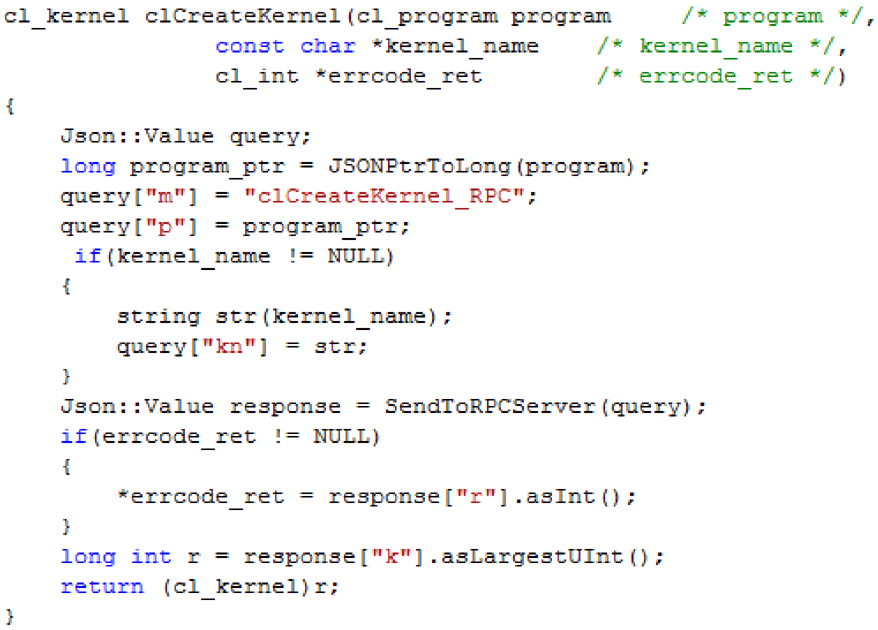
Şekil 5.10 : İstemci tarafında *clCreateCommandQueue* fonksiyonu.

Şekil 5.13 *clCreateKernel* fonksiyonunun dağıtım katmanında sunucu tarafında gerçeklenmesini göstermektedir. İstemci tarafından gönderilen program işaretçisi ve çekirdek adı parametreleri ile aynı isimli OpenCL fonksiyonu çağrısı yürütülüp dönüş parametreleri istemciye yollanmaktadır.



```
bool CLJsonRpc::clCreateCommandQueue_RPC(const Json::Value& root, Json::Value& response)
{
    int hostID = root["hi"].asInt();
    //std::cout << "Receive query: " << root << std::endl;
    cl_context context = (cl_context)root["c"].asLargestUInt();
    long a = root["d"].asLargestInt();
    cl_device_id device = (cl_device_id)root["d"].asLargestUInt();
    cl_command_queue_properties properties = (cl_command_queue_properties)root["p"].asLargestUInt();
    cl_int errNum;
    cl_command_queue *commandQueue = new cl_command_queue[1];
    *commandQueue = clCreateCommandQueue(context, device, properties, &errNum);
    response["r"] = errNum;
    long l_commandQueue = JSONPtrToLong(*commandQueue);
    response["cq"] = l_commandQueue;
    return true;
}
```

Şekil 5.11 : Sunucu tarafında *clCreateCommandQueue* fonksiyonu.

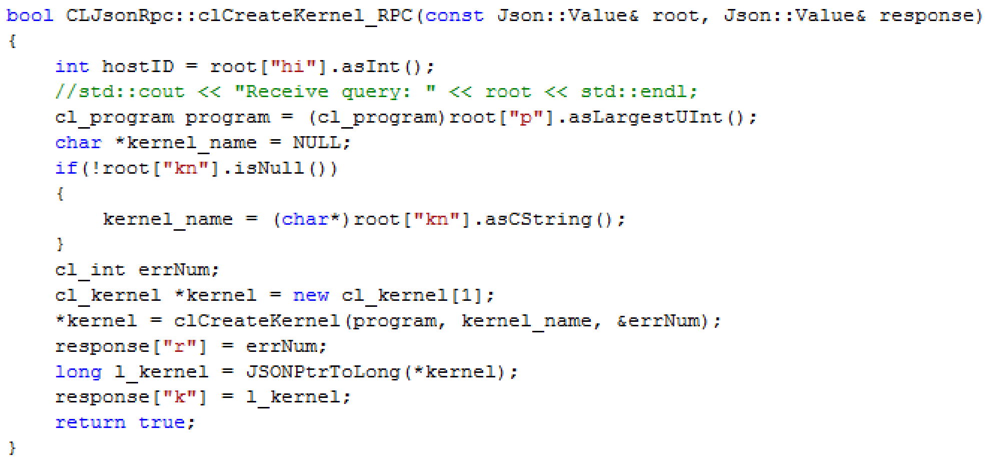


```
cl_kernel clCreateKernel(cl_program program     /* program */,
               const char *kernel_name    /* kernel_name */,
               cl_int *errcode_ret        /* errcode_ret */)
{
    Json::Value query;
    long program_ptr = JSONPtrToLong(program);
    query["m"] = "clCreateKernel_RPC";
    query["p"] = program_ptr;
     if(kernel_name != NULL)
    {
        string str(kernel_name);
        query["kn"] = str;
    }
    Json::Value response = SendToRPCServer(query);
    if(errcode_ret != NULL)
    {
        *errcode_ret = response["r"].asInt();
    }
    long int r = response["k"].asLargestUInt();
    return (cl_kernel)r;
}
```

Şekil 5.12 : İstemci tarafında *clCreateKernel* fonksiyonu.

Şekil 5.14 *clEnqueueReadBuffer* fonksiyonunun dağıtım katmanında istemci tarafında gerçeklenmesini göstermektedir. Komut kuyruğu işaretçisi, bellek alanı işaretçisi, blok okuma bayrağı, kayıklık (offset) değeri, okuma sonucunun yazılacağı alana ait işaretçi ve okunacak veri boyutu parametreleri sunucuya gönderilip cevap olarak dönen dizi değerleri ev sahibi programa yollanmaktadır.

Şekil 5.15 *clEnqueueReadBuffer* fonksiyonunun dağıtım katmanında sunucu tarafında gerçeklenmesini göstermektedir. İstemci tarafından gönderilen parametreler ile aynı isimli OpenCL fonksiyonu çağrısı yürütülüp dönüş parametreleri istemciye yollanmaktadır.

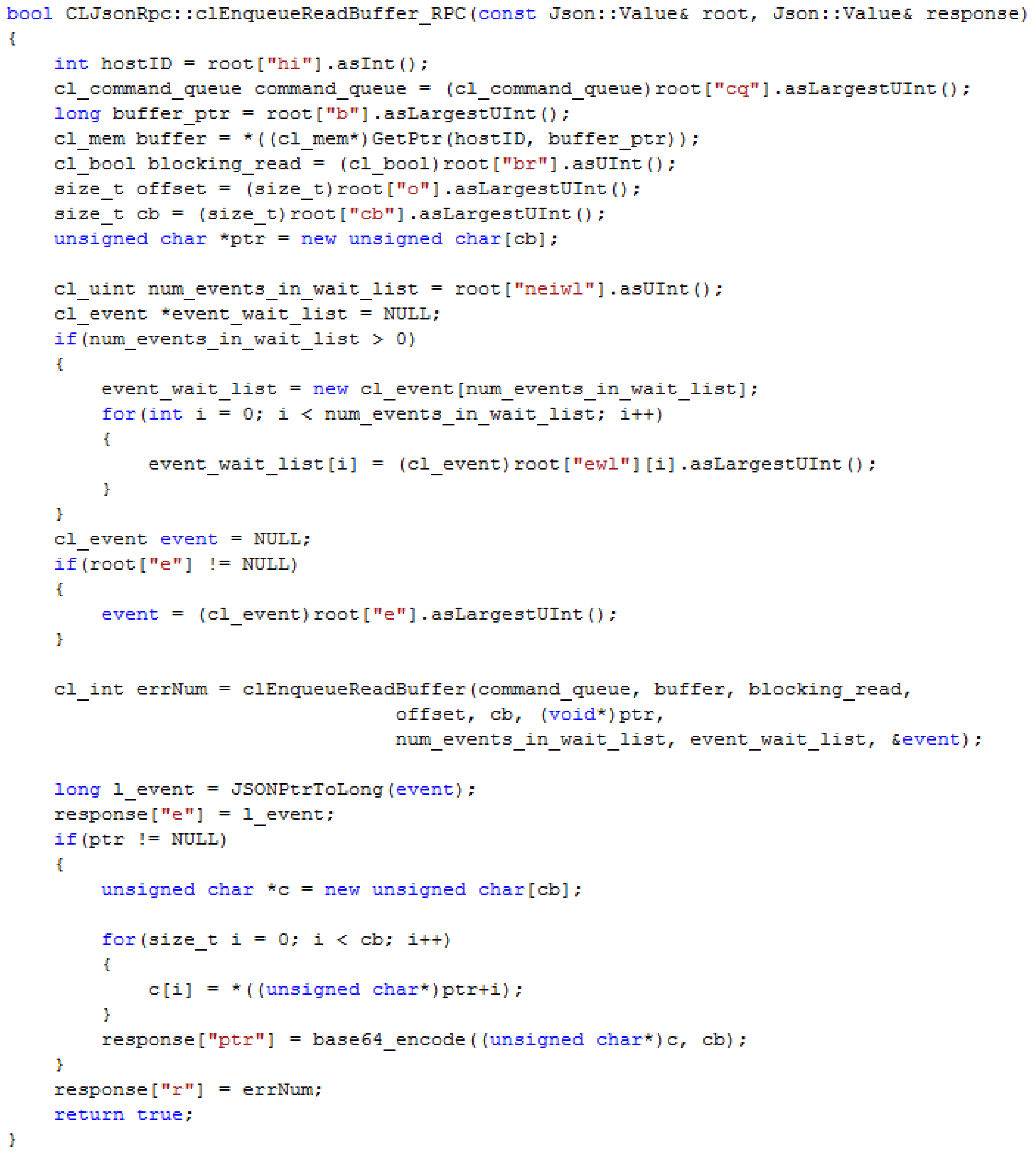


```
bool CLJsonRpc::clCreateKernel_RPC(const Json::Value& root, Json::Value& response)
{
    int hostID = root["hi"].asInt();
    //std::cout << "Receive query: " << root << std::endl;
    cl_program program = (cl_program)root["p"].asLargestUInt();
    char *kernel_name = NULL;
    if(!root["kn"].isNull())
    {
        kernel_name = (char*)root["kn"].asCString();
    }
    cl_int errNum;
    cl_kernel *kernel = new cl_kernel[1];
    *kernel = clCreateKernel(program, kernel_name, &errNum);
    response["r"] = errNum;
    long l_kernel = JSONPtrToLong(*kernel);
    response["k"] = l_kernel;
    return true;
}
```

Şekil 5.13 : Sunucu tarafında *clCreateKernel* fonksiyonu.

```
cl_int clEnqueueReadBuffer(cl_command_queue command_queue   /* command_queue */,
                    cl_mem buffer             /* buffer */,
                    cl_bool blocking_read            /* blocking_read */,
                    size_t offset             /* offset */,
                    size_t cb             /* cb */,
                    void *ptr              /* ptr */,
                    cl_uint num_events_in_wait_list            /* num_events_in_wait_list */,
                    const cl_event *event_wait_list    /* event_wait_list */,
                    cl_event *event          /* event */)
{
    Json::Value query;
    query["m"] = "clEnqueueReadBuffer_RPC";
    long command_queue_ptr = JSONPtrToLong(command_queue);
    query["cq"] = command_queue_ptr;
    long buffer_ptr = JSONPtrToLong(buffer);
    query["b"] = buffer_ptr;
    query["br"] = blocking_read;
    query["o"] = offset;
    query["cb"] = cb;
    query["neiwl"] = num_events_in_wait_list;
    for(int i = 0; i < num_events_in_wait_list; i++)
    {
        query["ewl"].append(event_wait_list[i]);
    }
    if(event != NULL)
    {
        long event_ptr = JSONPtrToLong(*event);
        query["e"] = event_ptr;
    }
    Json::Value response = SendToRPCServer(query);
    long l_event = response["e"].asLargestUInt();
    if(event != NULL)
    {
        *event = (cl_event)l_event;
    }
    if(cb > 0 && !response["ptr"].isNull())
    {
        std::string ptr_str = response["ptr"].asCString();
        std::string ptr_str_dec = base64_decode(ptr_str);
        for(long i = 0; i < cb; i++)
        {
            ((unsigned char*)ptr)[i] = ptr_str_dec[i];
        }
    }
    return response["r"].asInt();
}
```

Şekil 5.14 : İstemci tarafında *clEnqueueReadBuffer* fonksiyonu.

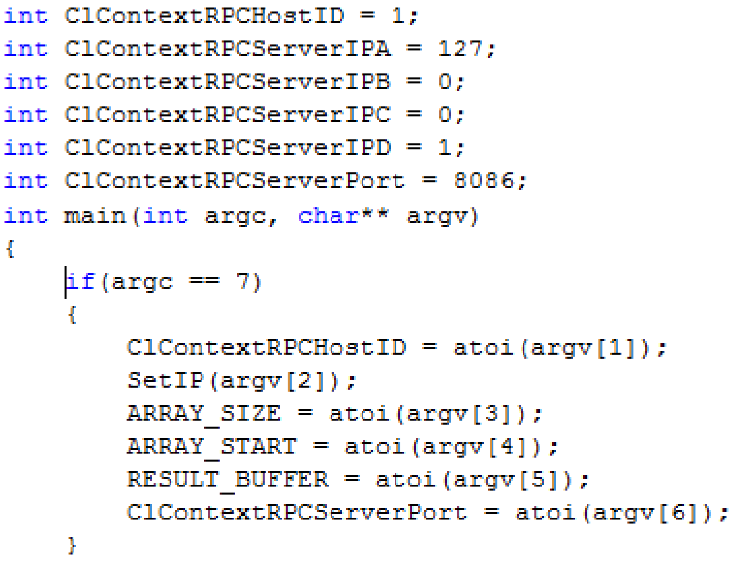


```
bool CLJsonRpc::clEnqueueReadBuffer_RPC(const Json::Value& root, Json::Value& response)
{
    int hostID = root["hi"].asInt();
    cl_command_queue command_queue = (cl_command_queue)root["cq"].asLargestUInt();
    long buffer_ptr = root["b"].asLargestUInt();
    cl_mem buffer = *((cl_mem*)GetPtr(hostID, buffer_ptr));
    cl_bool blocking_read = (cl_bool)root["br"].asUInt();
    size_t offset = (size_t)root["o"].asLargestUInt();
    size_t cb = (size_t)root["cb"].asLargestUInt();
    unsigned char *ptr = new unsigned char[cb];

    cl_uint num_events_in_wait_list = root["neiwl"].asUInt();
    cl_event *event_wait_list = NULL;
    if(num_events_in_wait_list > 0)
    {
        event_wait_list = new cl_event[num_events_in_wait_list];
        for(int i = 0; i < num_events_in_wait_list; i++)
        {
            event_wait_list[i] = (cl_event)root["ewl"][i].asLargestUInt();
        }
    }
    cl_event event = NULL;
    if(root["e"] != NULL)
    {
        event = (cl_event)root["e"].asLargestUInt();
    }

    cl_int errNum = clEnqueueReadBuffer(command_queue, buffer, blocking_read,
                             offset, cb, (void*)ptr,
                             num_events_in_wait_list, event_wait_list, &event);

    long l_event = JSONPtrToLong(event);
    response["e"] = l_event;
    if(ptr != NULL)
    {
        unsigned char *c = new unsigned char[cb];

        for(size_t i = 0; i < cb; i++)
        {
            c[i] = *((unsigned char*)ptr+i);
        }
        response["ptr"] = base64_encode((unsigned char*)c, cb);
    }
    response["r"] = errNum;
    return true;
}
```

Şekil 5.15 : Sunucu tarafında *clEnqueueReadBuffer* fonksiyonu.

5.7 Native OpenCL ile Distributed OpenCL Ev Sahibi Programları Arasındaki Farklar

Native OpenCL programlarını Distributed OpenCL çatısı ile bir istemci bir sunucu şeklinde çalıştırırken ev sahibi programa ait bir kimlik numarası (ID), sunucu IP adresi ve iletişimin kurulacağı port numarası parametreleri sağlanmaktadır. Şekil 5.16’da görüldüğü üzere bu parametreler ev sahibi programa komut satırı parametreleri olarak da verilebilmektedir. Birden fazla hesaplama düğümüyle paralel olarak çalışabilmek için, birden fazla eşlenik ev sahibi program çoklu iplik yapısı şeklinde verinin belli bölümleri üzerinde farklı sunucularla çalıştırılır. *ARRAY_SIZE,* *ARRAY_START* ve *RESULT_BUFFER* parametreleri verinin hangi bölümünün sunucuya gönderileceğini ve sonuç değerlerinin belleğin hangi bölgesine yazılacağını belirlemektedir.



```
int ClContextRPCHostID = 1;
int ClContextRPCServerIPA = 127;
int ClContextRPCServerIPB = 0;
int ClContextRPCServerIPC = 0;
int ClContextRPCServerIPD = 1;
int ClContextRPCServerPort = 8086;
int main(int argc, char** argv)
{
    if(argc == 7)
    {
        ClContextRPCHostID = atoi(argv[1]);
        SetIP(argv[2]);
        ARRAY_SIZE = atoi(argv[3]);
        ARRAY_START = atoi(argv[4]);
        RESULT_BUFFER = atoi(argv[5]);
        ClContextRPCServerPort = atoi(argv[6]);
    }
```

Şekil 5.16 : Distributed OpenCL ev sahibi programda ek parametreler.

5.8 Yapılan Testler ve Test Sonuçları

Distributed OpenCL çatısının performansını ölçmek için 3 farklı grupta testler yapılmıştır. Bunlar:

- Ek yük (overhead) testleri: Bu testlerde Doğal OpenCL (NCL - Native OpenCL) ile Distributed OpenCL arasındaki performans farkı; istemci program ile sunucu program aynı makinede çalışıyorken (DCL Yerel) ve farklı makinelerde çalışıyorken (DCL Uzak) süre ölçümleri yapılmıştır.

- Paralel yürütme testleri: Bu testlerde Distributed OpenCL çatısında hesaplama düğümü sayısı arttırıp paralel olarak çalıştırılarak hesaplama aygıtı başına düşen yük miktarı azaltılmış ve performansta elde edilen iyileşme gözlenmiştir. İstemci program bir, iki ve dört uzak sunucu ile çalıştırılarak işlem süreleri ölçüşmüştür.

- Çekirdek döngüsü testleri: Bu testlerde hesap yoğun işlemlerin benzetimi amacıyla çekirdek fonksiyonu içerisindeki işlemler döngüye sokulup birden fazla kez çalıştırılmıştır. Bu koşullar altında iki sistemin performansı ölçülmüştür.

Test işlemleri için vektör toplama, 2 boyutlu konvolusyion ve *N-Body* benzetimi çekirdek fonksiyonlarıyla bu fonksiyonları yürüten ev sahibi programlar kullanılmıştır.

Testler esnasında Native OpenCL ile Distributed OpenCL çatısı arasındaki performans farkındaki en belirleyici etkenin istemci ve sunucular arasında iletilen verinin boyutu olduğu görülmüştür. Dolayısıyla üç uygulama için de uygulamalar aynı veri miktarı ile çalıştırıldığında çalışma süreleri birbirine çok yakındır. Bu yüzden burada yalnızca vektör toplama uygulamasına ait sonuçlar sunulmaktadır.

Test sonuçları yürütme süreleri (execution time) göz önüne alınarak sunulmuş ve karşılaştırılmıştır. Testler 32'den 1M'ye kadar değişen iş parçacığı sayısıyla gerçekleştirilmiştir. Vektör toplama uygulaması iki *float* tipindeki diziyi toplamakta ve sonucu 3. Bir *float* dizisine yazmaktadır. Uygulamada istemci ve sunucular arasında aktarılan giriş verisi miktarı denklem **(5.1)** ile açıklanabilir.

$$V.B. = 3 \; x \; sizeof(float) \; x \; \text{iş parçacığı sayısı} \qquad \textbf{(5.1)}$$

$$V.B. = 12 \; x \; \text{iş parçacığı sayısı}$$

Her test 10 kez çalıştırılmış ve sonuçlarda bu testlerin medyanı gösterilmektedir. Uzak bilgisayardaki testler için istemci ve sunucular 100 Mbit Ethernet (LAN) bağlantısı kullanılarak bağlanmıştır. Çizelge 5.1 testlerde kullanılan bilgisayarların özelliklerini göstermektedir. GPU'larla ilgili detaylı özellik bilgileri üreticilerin sayfalarında bulunmaktadır.

5.8.1 Ek yük testleri

Ek yük testlerinde Distributed OpenCL çatısındaki iletişim maliyetinden kaynaklanan ek yükün işlem performansına etkisi gözlemlenmiştir. Aşağıdaki testler gerçekleştirilmiştir:

Çizelge 5.1 : Testlerde kullanılan bilgisayarların özellikleri.

Bilg. ID	İşletim Sistemi	CPU	Bellek	GPU
A	Windows 7 64 bit	Intel Core i7	4 GB DDR 3	NVIDIA 525 M
B	Windows 7 64 bit	Intel Core i7	4 GB DDR 3	ATI RADEON HD 6370 M
C	Ubuntu Linux 64 bit	Intel Core 2 Duo	4 GB DDR 3	NVIDIA 240M
D	Ubuntu Linux 64 bit	Intel Core i7	4 GB DDR 3	NVIDIA 540 M
E	Windows 7 64 bit	Intel Core i7	8 GB DDR 3	NVIDIA 555 M
F	Windows 7 64 bit	Intel Core i5	3 GB DDR 3	NVIDIA 520 M

- Test 1: Test uygulaması bilgisayarlarda Native OpenCL çatısı ile test edilmiştir. Bilgisayarlardan A, B ve C için sonuçlar Şekil 5.17'de gösterilmektedir (NCL) (Şekil 5.17'de; E.S. ID – ev sahibi bilgisayar IDsi, GPU ID – hesaplamaların yapıldığı bilgisayar IDsi, V.B. – iletilen veri boyutu, Y.Z. – yürütme zamanını göstermektedir).

- Test 2: Test uygulaması Distributed OpenCL çatısı ile istemci ve sunucu program aynı bilgisayarda çalışacak şekilde test edilmiştir. A, C ve D için sonuçlar Şekil 5.17'de gösterilmektedir (DCL Yerel).

- Test 3: Test uygulaması Distibuted OpenCL çatısı ile istemci ve sunucu program farklı bilgisayarlarda çalışacak şekilde test edilmiştir. Testler makinelerin ikili kombinasyonlarıyla gerçekleştirilmiştir. C (istemci) - A (sunucu), A (istemci) – E (sunucu), A (istemci) – C (sunucu) ve C (istemci) – D (sunucu) bilgisayarlarındaki sonuçlar Şekil 5.17'de gösterilmektedir (DCL Uzak).

5.8.2 Paralel yürütme testleri

Paralel yürütme testleri, yük paralel çalışabilen birden fazla hesaplama aygıtına dağıtıldığı durumlarda Distributed OpenCL çatısının performansını gözlemlemek için gerçekleştirilmiştir. Yapılan testler şu şekildedir:

- Test 1: Test uygulaması bir istemci ve bir sunucu program ile bilgisayarların ikili kombinasyonlarında test edilmiştir. C (istemci) - A (sunucu), A (istemci) – E (sunucu), A (istemci) – C (sunucu) ve C (istemci) – D (sunucu) bilgisayarlarındaki sonuçlar Şekil 5.18’de gösterilmektedir (DCL Uzak 1 bilg.) (Şekil 5.18’de E.S. ID – ev sahibi bilgisayar IDsi, GPU ID – hesaplamaların yapıldığı bilgisayar IDsi, V.B. – iletilen veri boyutu, Y.Z. – yürütme zamanını göstermektedir).

- Test 2: Test uygulaması bir istemci ve paralel çalışan iki sunucu program ile bilgisayarların üçlü kombinasyonlarında çalıştırılmıştır. C (istemci) – A ve E (sunucu), A (istemci) – D ve E (sunucu) bilgisayarlarındaki sonuçlar Şekil 5.18’de gösterilmektedir (DCL Uzak 2 bilg.).
- Test 3: Test uygulaması bilgisayarların beşli kombinasyonlarında bir istemci ve dört sunucu programla çalıştırılmıştır. C (istemci) – A, B, D ve E (sunucu); F (istemci) – A, B, D ve E (sunucu) bilgisayarlarındaki sonuçlar Şekil 5.18’de gösterilmektedir (DCL Uzak 4 bilg.).

5.8.3 Çekirdek döngüsü testleri

Distributed OpenCL çatısının hesap yoğun işlemlerdeki performansını değerlendirmek amacıyla çekirdek fonksiyonu içerisindeki işlemler döngüye sokulup birden fazla kez çalıştırılmıştır. Bu koşullar altında iki sistemin performansı ölçülmüştür. Şekil 5.19’da test uygulamasının NCL, DCL Yerel, DCL Uzak 1 bilgisayar, DCL Uzak 2 bilgisayar, DCL Uzak 4 bilgisayarda çalıştırıldığındaki yürütme süreleri farklı veri boyutlarında ve çekirdek fonksiyondaki farklı döngü sayıları için gösterilmektedir (Şekil 5.19’da E.S. ID – ev Sahibi bilgisayar IDsi, GPU ID – hesaplamaların yapıldığı bilgisayar IDsi, V.B. – iletilen veri boyutu, Y.Z. – yürütme zamanını göstermektedir). Test uygulaması 2048, 4096 ve 8192 iş parçacığı ile çekirdek fonksiyonundaki toplama işlemi 32, 1024, 16384 ve 65536 kez döngüye sokularak yürütme süreleri ölçülmüştür.

5.8.4 Sonuçların değerlendirilmesi

İlk test grubu olan ek yük testlerinden Test 1 için aşağıdaki çıkarımlar yapılmaktadır:

- GPU mimarisinin paralel işlemeyi sağlaması nedeniyle iş parçacığı sayısı ile yürütme süresi arasında yaklaşık doğrusal bir ilişki bulunmamaktadır.
- Native OpenCL’de NVIDIA GPU (NCL A) ile eş değer bir ATI GPU (NCL B) arasında çok fazla bir performans farkı görülmemektedir.

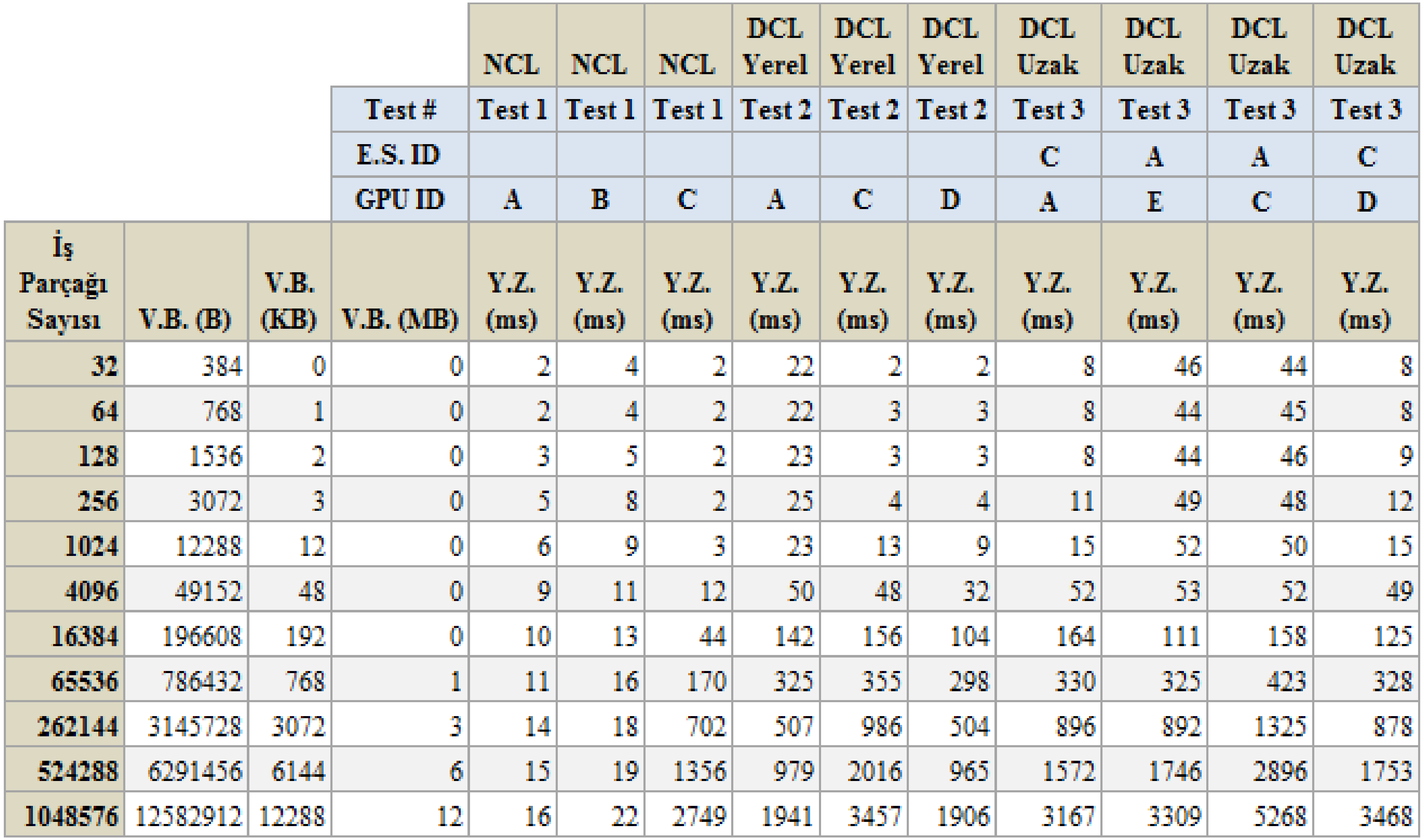
- Bu karşın, daha eski model bir NVIDIA GPU (NCL C) ile çekirdek sayısı ve tampon boyutu (buffer size) daha küçük olduğu için OpenCL yürütme süresi iş parçacığı sayısının artmasıyla beraber hızla artmaktadır.

Test 2’den çıkarılan sonuçlar şu şekildedir:

- NCL ile DCL Yerel’i karşılaştırdığımızda yürütme süresi iletişim yükünden dolayı artmaktadır. İletişim yükü özellikle *clEnqueWriteBuffer, clEnqueueReadBuffer, clCreateBuffer* gibi OpenCL aygıtına giriş verisinin gönderildiği veya sonuç verisinin okunduğu fonksiyonlarda etkilidir. İletişim yükü büyük miktardaki veri boyutlarında, iletilen veri boyutunun artışıyla birlikte doğrusala yakın olarak artar.
- İşletim sistemi yapılandırmasına bağlı olarak TCP veri aktarım süresi belli bir veri boyutundan sonra daha hızlı artmaktadır, çünkü aktarılan veri tek bir TCP paketine sığmamaktadır.
- Küçük veri boyutları için Linux işletim sistemine sahip bir makineden Linux veya Windows işletim sistemine sahip bir makineye veri iletim hızı işletim sisteminin karakteristiklerinden dolayı daha yüksektir. Veri boyutu arttıkça aradaki belirgin bir fark görülmemektedir.
- İletişim yükü GPU modelinden bağımsız iken, iletişim yükünün genel hesaplama süresine oransal olarak etkisi değişmektedir. Çünkü sunucularda yürütülen OpenCL fonksiyonlarının süresi oldukça değişkendir. NCL A – DCL Yerel A ile NCL C – DCL Yerel C arasındaki performans farkları karşılaştırıldığında bu durum açık bir şekilde görülmektedir.

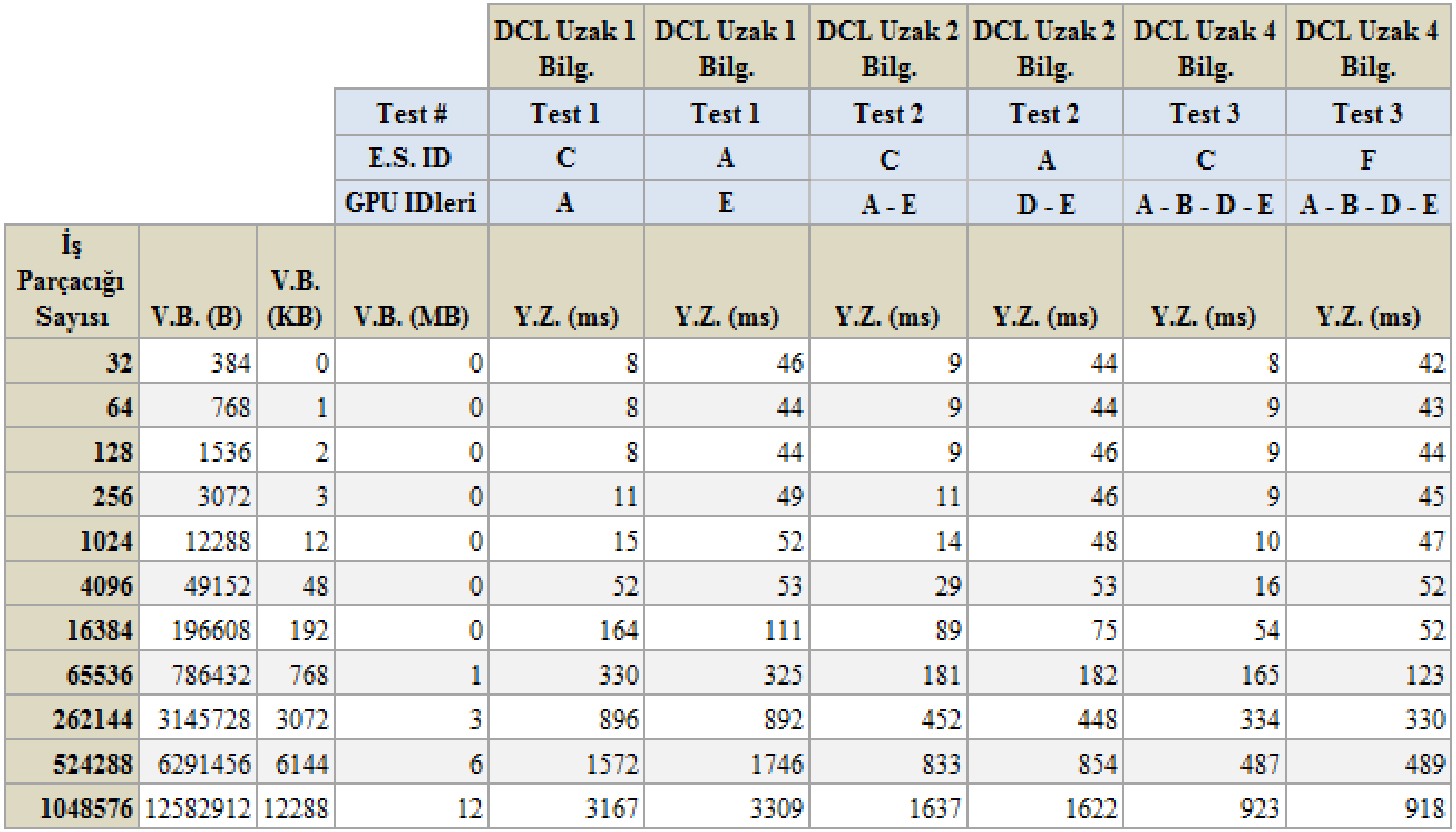
Test 3 sonuçlarına göre DCL Yerel ile DCL Uzak karşılaştırıldığında 100 Mbit Ethernet bağlantısı ile yürütme süresinin 1.5 – 2 katına çıktığı görülmektedir.

İkinci test grubu olan paralel yürütme testleri iş yükünü paralel olarak çalışan birden fazla hesaplama düğümüne dağıtmanın performansı arttıracağını göstermektedir. Şekil 5.20’de yürütme süresinin hesaplama düğümü sayısındaki artışla birlikte neredeyse aynı oranda azaldığı görülmektedir. Veri boyutu küçük iken yürütme süresi hesaplama düğümü sayısındaki artmadan çok etkilenmez. Çünkü bu kadar veri boyutlarında giriş – çıkış verisi transfer işlemlerinin yanısıra diğer OpenCL fonksiyon çağrıları da yürütme zamanını etkilemektedir.



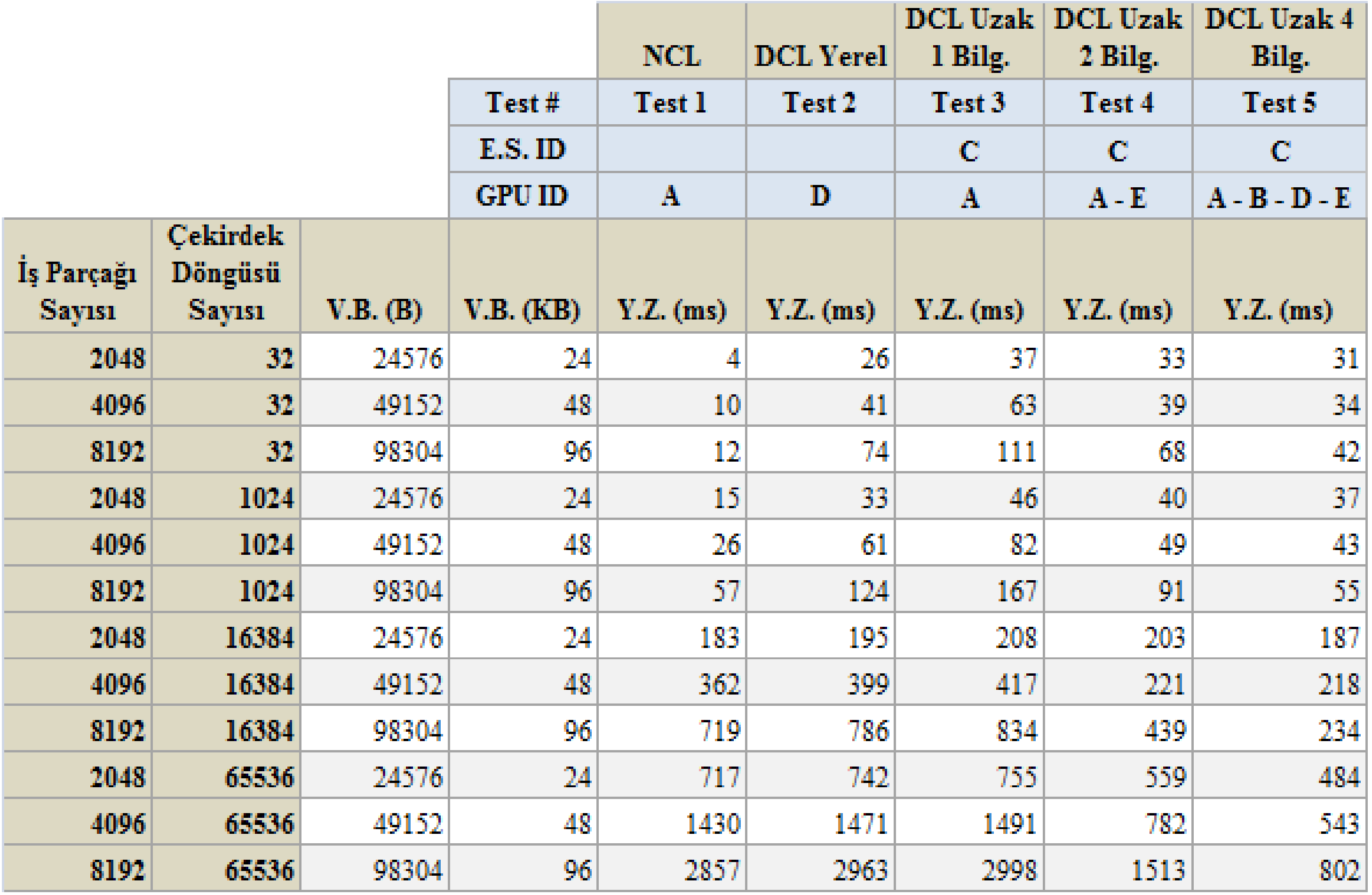
				NCL	NCL	NCL	DCL Yerel	DCL Yerel	DCL Yerel	DCL Uzak	DCL Uzak	DCL Uzak	DCL Uzak
			Test #	Test 1	Test 1	Test 1	Test 2	Test 2	Test 2	Test 3	Test 3	Test 3	Test 3
			E.S. ID							C	A	A	C
			GPU ID	A	B	C	A	C	D	A	E	C	D
İş Parçağı Sayısı	V.B. (B)	V.B. (KB)	V.B. (MB)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)
32	384	0	0	2	4	2	22	2	2	8	46	44	8
64	768	1	0	2	4	2	22	3	3	8	44	45	8
128	1536	2	0	3	5	2	23	3	3	8	44	46	9
256	3072	3	0	5	8	2	25	4	4	11	49	48	12
1024	12288	12	0	6	9	3	23	13	9	15	52	50	15
4096	49152	48	0	9	11	12	50	48	32	52	53	52	49
16384	196608	192	0	10	13	44	142	156	104	164	111	158	125
65536	786432	768	1	11	16	170	325	355	298	330	325	423	328
262144	3145728	3072	3	14	18	702	507	986	504	896	892	1325	878
524288	6291456	6144	6	15	19	1356	979	2016	965	1572	1746	2896	1753
1048576	12582912	12288	12	16	22	2749	1941	3457	1906	3167	3309	5268	3468

Şekil 5.17 : Ek yük testleri sonuçları.



				DCL Uzak 1 Bilg.	DCL Uzak 1 Bilg.	DCL Uzak 2 Bilg.	DCL Uzak 2 Bilg.	DCL Uzak 4 Bilg.	DCL Uzak 4 Bilg.
			Test #	**Test 1**	**Test 1**	**Test 2**	**Test 2**	**Test 3**	**Test 3**
			E.S. ID	**C**	**A**	**C**	**A**	**C**	**F**
			GPU IDleri	**A**	**E**	**A - E**	**D - E**	**A - B - D - E**	**A - B - D - E**
İş Parçacığı Sayısı	**V.B. (B)**	**V.B. (KB)**	**V.B. (MB)**	**Y.Z. (ms)**	**Y.Z. (ms)**	**Y.Z. (ms)**	**Y.Z. (ms)**	**Y.Z. (ms)**	**Y.Z. (ms)**
32	384	0	0	8	46	9	44	8	42
64	768	1	0	8	44	9	44	9	43
128	1536	2	0	8	44	9	46	9	44
256	3072	3	0	11	49	11	46	9	45
1024	12288	12	0	15	52	14	48	10	47
4096	49152	48	0	52	53	29	53	16	52
16384	196608	192	0	164	111	89	75	54	52
65536	786432	768	1	330	325	181	182	165	123
262144	3145728	3072	3	896	892	452	448	334	330
524288	6291456	6144	6	1572	1746	833	854	487	489
1048576	12582912	12288	12	3167	3309	1637	1622	923	918

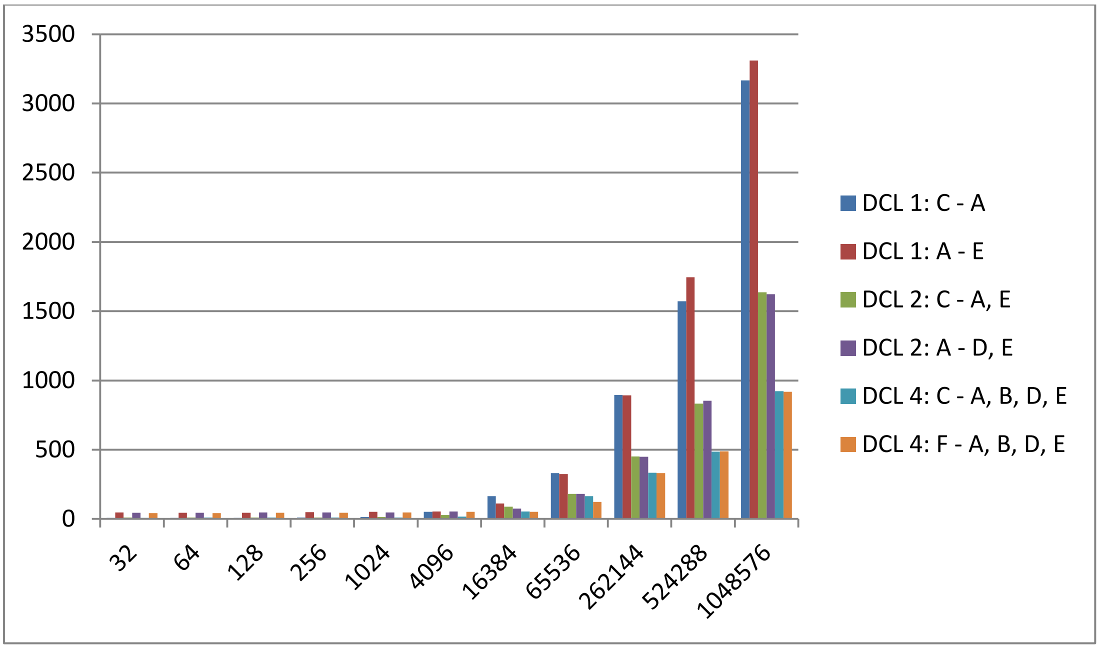
Şekil 5.18 : Paralel yürütme testleri sonuçları.



				NCL	DCL Yerel	DCL Uzak 1 Bilg.	DCL Uzak 2 Bilg.	DCL Uzak 4 Bilg.
			Test #	Test 1	Test 2	Test 3	Test 4	Test 5
			E.S. ID			C	C	C
			GPU ID	A	D	A	A - E	A - B - D - E
İş Parçağı Sayısı	Çekirdek Döngüsü Sayısı	V.B. (B)	V.B. (KB)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)	Y.Z. (ms)
2048	**32**	24576	24	4	26	37	33	31
4096	**32**	49152	48	10	41	63	39	34
8192	**32**	98304	96	12	74	111	68	42
2048	**1024**	24576	24	15	33	46	40	37
4096	**1024**	49152	48	26	61	82	49	43
8192	**1024**	98304	96	57	124	167	91	55
2048	**16384**	24576	24	183	195	208	203	187
4096	**16384**	49152	48	362	399	417	221	218
8192	**16384**	98304	96	719	786	834	439	234
2048	**65536**	24576	24	717	742	755	559	484
4096	**65536**	49152	48	1430	1471	1491	782	543
8192	**65536**	98304	96	2857	2963	2998	1513	802

Şekil 5.19 : Çekirdek döngüsü testleri sonuçları.

Üçüncü test grubu olan çekirdek döngüsü testlerinin sonuçlarına göre Native OpenCL'deki yürütme süresi döngü sayısının artmasıyla birlikte logaritmiğe yakın bir biçimde artmaktadır. Hesap yoğun çekirdek fonksiyonlarında sistemlerin davranışını gözlemlemek için yapılan bu testlerde, iş yükü Distributed OpenCL ile 4 GPU'ya dağıtıldığında döngü sayısı arttıkça Native OpenCL'e göre daha iyi performans elde edilmiştir.



Şekil 5.20 : Paralel yürütme testleri sonuç grafiği.

Tüm test sonuçları göz önüne alındığında Distributed OpenCL çatısında yürütme süresine belirleyen etmenler denklem **(5.2)** ve **(5.3)** ile açıklanabilir. Denklem **(5.2)**'de tek bir bilgisayarla, **(5.3)**'te ise birden fazla bilgisayarla çalışan Distributed OpenCL sistemlerinin yürütme sürüleri denklemleri gösterilmektedir.

$$Y.S. = \text{TCP Bağlantı Kurma Süresi} + \text{OpenCL Hesaplama Süresi} + \text{Veri Transferi Süresi} \quad \mathbf{(5.2)}$$

$$Y.S. = \text{Düğüm Sayısı x TCP Bağlantı Açma Süresi} + \max(\text{OpenCL Hesaplama Süresi} + \text{Veri Transferi Süresi}) \quad \mathbf{(5.3)}$$

Bu denklemlerde TCP bağlantı kurma süresi istemci bilgisayar ile sunucu bilgisayarlar arasındaki TCP bağlantısının başlatılması için gereken süredir. Düğüm sayısı arttığında her bir düğüm için bu süre harcanır. OpenCL hesaplama süresi, hesaplama aygıtında OpenCL fonksiyonlarının çalışma süresidir. Veri miktarı ve çekirdek fonksiyonun hesap yoğunluğu bu süreyi belirleyen etmenlerdir. Veri transferi süresi ise giriş ve sonuç verilerinin istemci ve sunucular arasında aktarımı için gerekli süredir. Bu süreye veri miktarı ile birlikte işletim sisteminin iletişim yapısı, TCP yapılandırması, maksimum izin verilen TCP paket boyutu, kullanılan iletişim ağının hızı gibi etmenler de etki etmektedir.

5.8.5 Önceki çalışmalarla karşılaştırmalar

Tez çalışması esnasında yapılan literatür taramalarında *Many GPUs Package* (MPG)[19], *Virtual OpenCL Cluster Platform* (VCL) [20], *Remote CUDA* (rCUDA) [21], *Hybrid OpenCL* [22], *CLuMPI* [23] ve *GPU Clusters for High-Performance Computing* [24] gibi GPGPU üzerine yapılmış olan çalışmalara rastlanmıştır. Bu çalışmalar ya üreticiye (*vendor specific*) özel ya da işletim sistemi bağımlı gerçeklemeler sunmaktadırlar. Bu tez çalışmasında geliştirilmiş olan Distributed OpenCL çatısı GPGPU hesaplama için OpenCL’i ve iletişim tekniği olarak da JSON - RPC protokolünü kullandığı için hem üretici hem de işletim sistemi bağımsız olarak çalışabilmektedir.

6. SONUÇ

OpenCL; GPU, CPU ve diğer işlem birimlerinden oluşan heterojen ortamlarda GPGPU programlarını çalıştırmaya yarayan bir çatıdır. Sunmuş olduğu genel geçer API ile platform ve üretici bağımsızlığı sağlamaktadır. OpenCL standartlarının kabulünün gün geçtikçe artması ve önde gelen üreticilerin katkısı ile OpenCL'in kullanımı giderek artmaktadır.

OpenCL'in özellikleri ve mimarisi düşünüldüğünde OpenCL, hesaplama düğümlerinin ağ ölçeğinde dağıtılmasına izin veren bir yapıdadır. Bu tez çalışmasında ev sahibi uygulamaların çalıştığı istemciler ile OpenCL kullanan hesaplama aygıtlarının çalıştığı paralel olarak işlem yapabilen sunuculardan oluşan, istemci ve sunucular arası iletişimin JSON – RPC protokolü ile sağlandığı Distributed OpenCL çatısı geliştirilmiştir. Test sonuçlarından görüldüğü üzere çoklu GPU barındıran sunucular, *Infiniband* gibi düşük gecikmeli – yüksek hızlı ağlar kullanıldığında ve maksimum izin verilen TCP paket boyutu arttırılabildiğinde; üretici – işletim sistemi bağımsız, dağıtık, paralel çalışan, ölçeklenebilir ve genel hesaplama performansında artış sağlayan bir GPGPU sistemi gerçeklemek mümkündür.

KAYNAKLAR

[1] **Url-1** <*http://www.nvidia.com/cuda*>, alındığı tarih: 20.03.2012.

[2] **Url-2** <*http://graphics.stanford.edu/projects/brookgpu*>, alındığı tarih: 20.03.2012.

[3] **Url-3** <*http://msdn.microsoft.com/directx*>, alındığı tarih: 20.03.2012.

[4] **Url-4** <*http://www.khronos.org/opencl/*>, alındığı tarih: 20.03.2012.

[5] **Url-5** <*http:// json-rpc.org*>, alındığı tarih: 20.03.2012.

[6] **Lippert, A.** (2009). NVIDIA GPU Architecture for General Purpose Computing, 4.

[7] **Url-6** <*http://www.khronos.org/opengles/2_X/*>, alındığı tarih: 27.04.2012.

[8] **Url-7** <*http://www.viznet.ac.uk/reports/gpu/10*>, alındığı tarih: 28.04.2012.

[9] **Datar, A. ve Padhye, A.** (2005). Graphics Processing Unit Architecture With a Focus on NVIDIA GeForce 6800 GPU, 7.

[10] **Davis, U. ve Owens, J.** (2007). GPU Architecture Overview. SIGGRAPH2007, 10.

[11] **Lippert, A.** (2009). NVIDIA GPU Architecture for General Purpose Computing, 18.

[12] **Lippert, A.** (2009). NVIDIA GPU Architecture for General Purpose Computing, 20.

[13] **Url-8** <*http://developer.nvidia.com/cg-toolkit/*>, alındığı tarih: 22.04.2012.

[14] **Url-9** <*http://sourceforge.net/projects/amdctm/*>, alındığı tarih: 22.04.2012.

[15] **Munshi, A. (editör), Khoronos Group (2009)** The OpenCL Specification, 20.

[16] **Munshi, A. (editör), Khoronos Group (2009)** The OpenCL Specification, 22.

[17] **Munshi, A. (editör), Khoronos Group (2009)** The OpenCL Specification, 25.

[18] **Url-10** <*http://jsonrpc-cpp.sourceforge.net/*>, alındığı tarih: 23.03.2012.

[19] **Barak, A., Ben-Nun, T., Levy, E. ve Shiloh, A.** A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices.

[20] **Barak, A. ve Shiloh, A.** The Virtual OpenCL (VCL) Cluster Platform.

[21] **Url-11** <*http://www.rcuda.net/*>, alındığı tarih: 17.04.2012.

[22] **Aoki, R., Oikava, S., Tsuchiyama, R., Nakamura, T. (2010)** Improving Hybrid OpenCL Performance by High Speed Networks *2010 First IEEE international Conference on Networking and Computing.*

[23] **Url-12** <*http://clumpi.sourceforge.net*>, alındığı tarih: 19.04.2012.

[24] **Kindratenko, V.** GPU Clusters for High-Performance Computing.

ÖZGEÇMİŞ

Ad Soyad: Barış Eskikaya

Doğum Yeri ve Tarihi: 25 Ağustos 1987

E-Posta: eskikayab@itu.edu.tr

Lisans: İstanbul Teknik Üniversitesi, Bilgisayar Mühendisliği

Yüksek Lisans: İstanbul Teknik Üniversitesi, Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği

TEZDEN TÜRETİLEN YAYINLAR/SUNUMLAR

- Eskikaya B., Altılar D. T., 2012: Distributed OpenCL, Distributing OpenCL on Network Scale. *Third International Conference on Advanced Computing and Communication Technologies for High Performance Applications*, June 21-23, 2012 Cochin, Kerela, India.