T.C.
ÇANAKKALE ONSEKİZ MART ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

STAJ RAPORU

Öğrencinin Numarası : 160401037

Öğrencinin Adı Soyadı : Kadir ÇOLAK

Staj İsmi ve Kodu : BLM-3007 Staj I

Öğretim Elemanının Unvanı/Adı Soyadı : Prof . Dr İsmail KADAYIF

OpenCL (Open Computing Language)

KISALTMALAR

API : Application Programming Interface

CPU : Central Processing Unit

CUDA : Common Unified Device Architecture

OpenCL : Open Computing Language

GPU : Graphic Processing Unit

GPGPU : General Purpose programming on Graphic Processing Unit

GFLOPS/s : Giga Floating-Point Operations per Second

HPC : High Performance Computing

SIMD : Single Instruction Multiple Data

SPMD : Single Program Multiple Data

FPGA: Field Programmable Gate Array

ŞEKİL LİSTESİ

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 kullanılabilmekte 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) ,BrookGPU (Stanford Üniversitesi) , DirectCompute (Microsoft) 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) 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 çalışmada Grafik kartlarının yapıları ,OpenCL terimleri mimarileri çalışma şekli ve OpenCL programlama kuralları adımları ve örnekleri incelenmiştir.

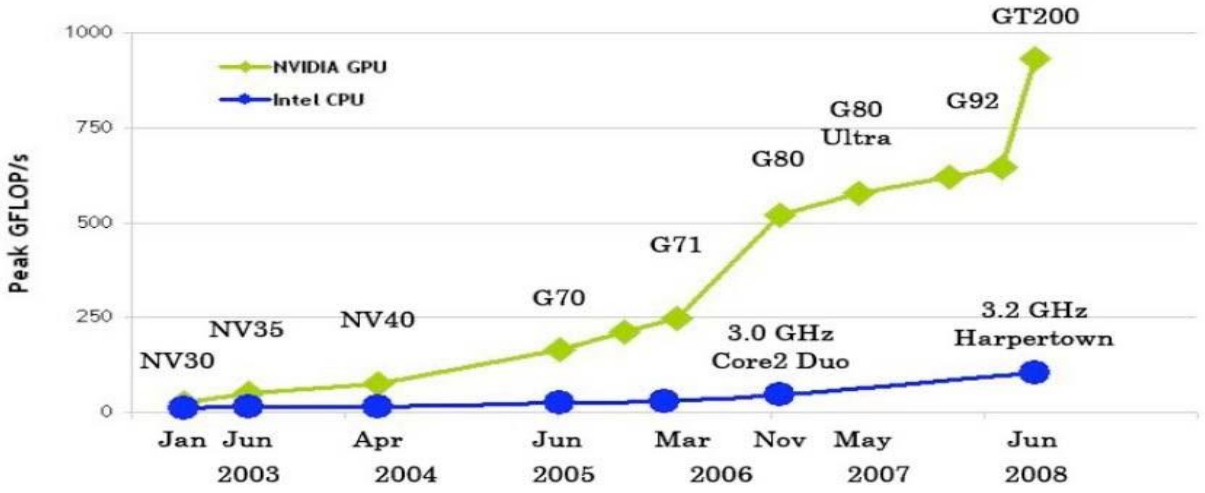
1.GRAFİK KARTLARI, GRAFİK İŞLEM BİRİMLERİ VE GPGPU KAVRAMI

1.1 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 gerçekleştiren 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.

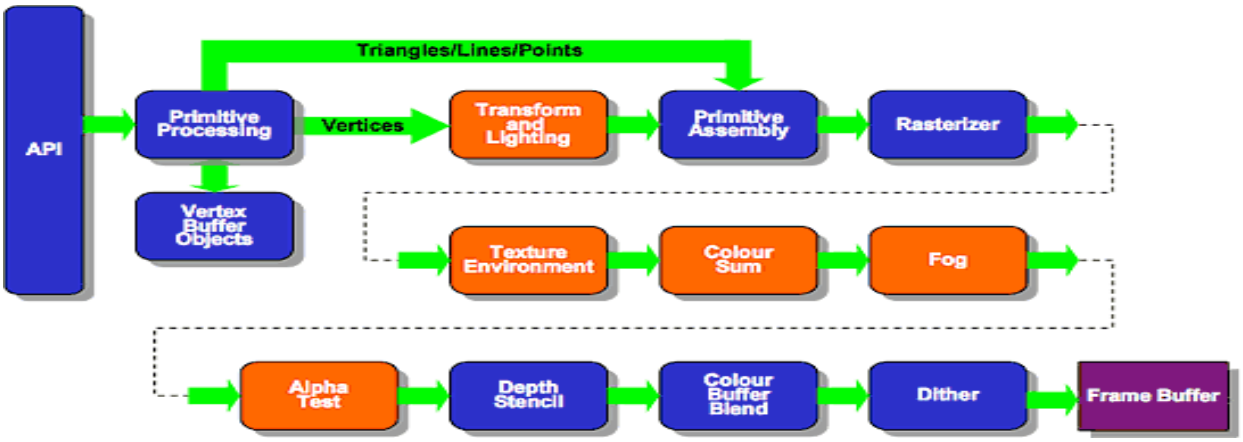
1.2. GRAFİK İŞLEMCİLERİNİN YAPISI

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 1.1'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.



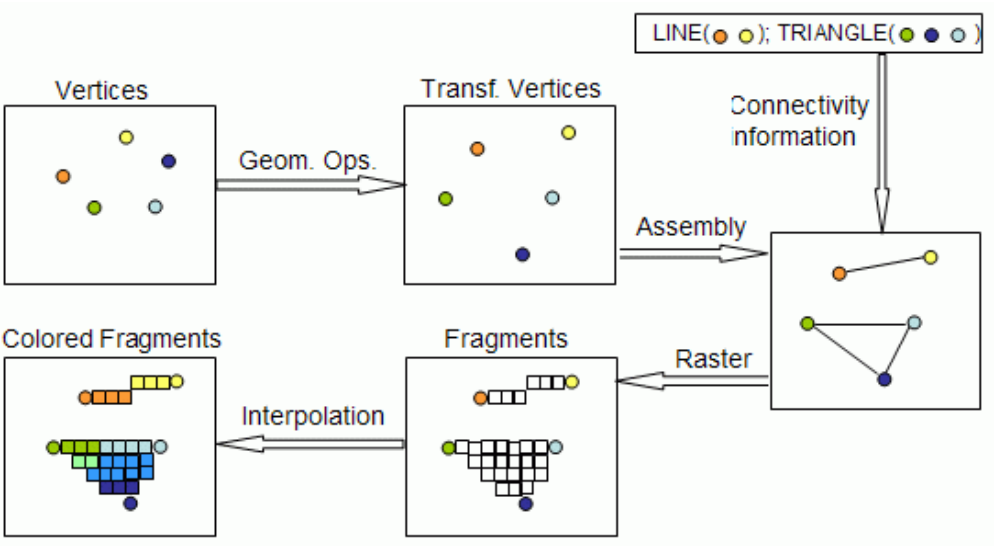
Şekil1. 1 **Intel CPU ve NVIDIA GPU'ların işlem hızı karşılaştırması**

Grafik işlemcilerini programlanabilmesi için öncelikle grafik işlemcilerin mimari yapısının bilinmesi gereklidir. Grafik işlemcilerin ilk çıkış noktası, CPU'ların hem normal işlemleri, hem de grafik işlemlerini beraber yaparken yeterli performans verememesidir. İlk üretilen grafik işlemciler sadece daha iyi grafik ortaya çıkarabilmek amacıyla üretilmiştir. İlk grafik işlemcilerde sadece belirli yöntemleri izleyen sabit bir grafik boru hattı bulunmaktaydı. Her çerçeve (frame) aynı boru hattından geçerek sonunda çerçeve tamponu (framebuffer) adı verilen birimde depolanmaktaydı. Buradan da çıkış aygıtı olan ekrana verilmekteydi. Grafik işlemcilerinin boru hattı Şekil 1.2' de gösterilmiştir

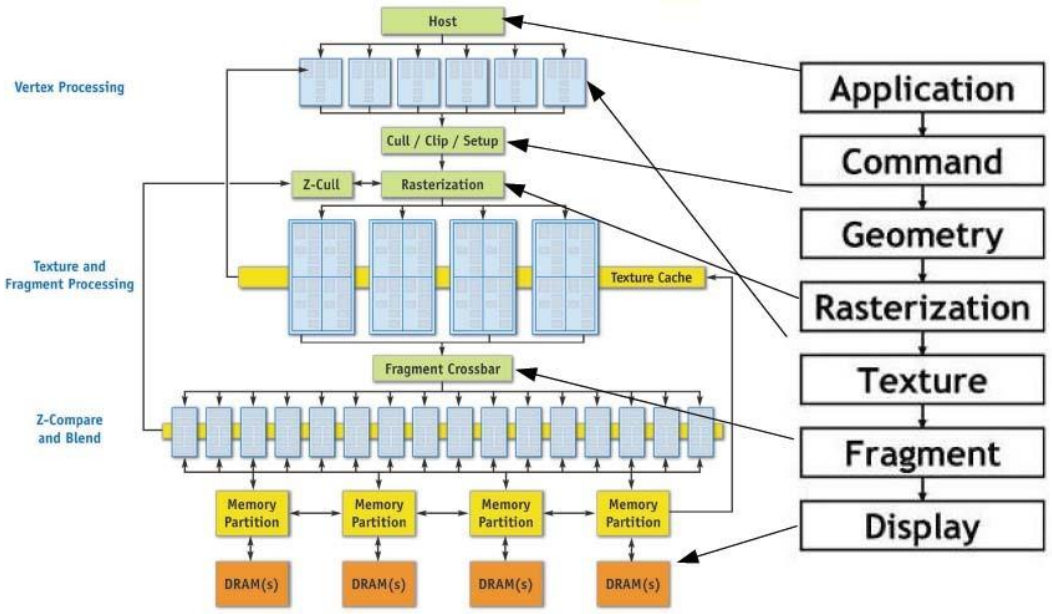


Şekil1. 2 Gpu işlem hattı mimarisi

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 1.2'de GPU işlem hattı mimarisi, Şekil 1.3'te işlem hattına giren verinin ne şekilde görüntüye dönüştürüldüğü, Şekil 1.4'te ise NVIDIA GeForce 6800 ekran kartına ait blok diyagramı üzerinde işlem hattının gerçeklenmesi gösterilmiştir. Şekil 1.2 ve Şekil 1.3'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.



Şekil1. 3 İşlem hattında veri dönüşümü



Şekil1. 4 NVIDIA GeForce 6800 blok diyagramı

Birleşik gölgelendiricilerin ortaya çıkması ile birlikte NVidia, genel amaçlı grafik işlemci mimarisini ve mimariye uygun programlama modeli olan CUDA (common unified device architecture) ile genel amaçlı grafik işlemcileri piyasaya sürmüştür. Genel amaçlı grafik işlemcilerin kullanımı sıklaştıktan sonra, Khronos Grubu programlama modelini standart haline getirmiş ve OpenCL (Open Computing Language) programlama modelini çıkarmıştır

1.3. Genel Amaçlı Grafik İşlemci Programlama Modeli

CUDA ve OpenCL grafik işlemcilerde genel amaçlı işlemler yapma imkanı sağlayan altyapılardan ilk ikisidir. CUDA 2006 yılında ve OpenCL de 2008 yılında piyasaya sürülmüştür. CUDA Nvidia firması tarafından geliştirilmiş ve yalnızca Nvidia grafik işlemcileri üzerinde çalışabilmektedir. Öte yandan Khronos Grup tarafından gereksinimleri ve platform tanımları yapılan OpenCL, telif hakkı kimseye ait olmayan, çapraz-platform uyumlu bir modeldir. Gereksinimleri gerçekleyen her üretici firma OpenCL desteği verebilmektedir. Bu sebeple OpenCL genel amaçlı işlemciler (CPU), grafik işlemciler (GPU), sinyal işleyiciler (DSP) ve alanda programlanabilir kapı dizileri (FPGA) gibi platformlarda destek verebilmektedir. Gömülü sistem platformları da OpenCL desteği bulunan platformlar arasına son yıllarda katılmıştır. OpenCL farklı konularda artıları olan platformları bir araya getirerek heterojen bir çalışma ortamı sağlayarak en yüksek başarıma ulaşabilir.

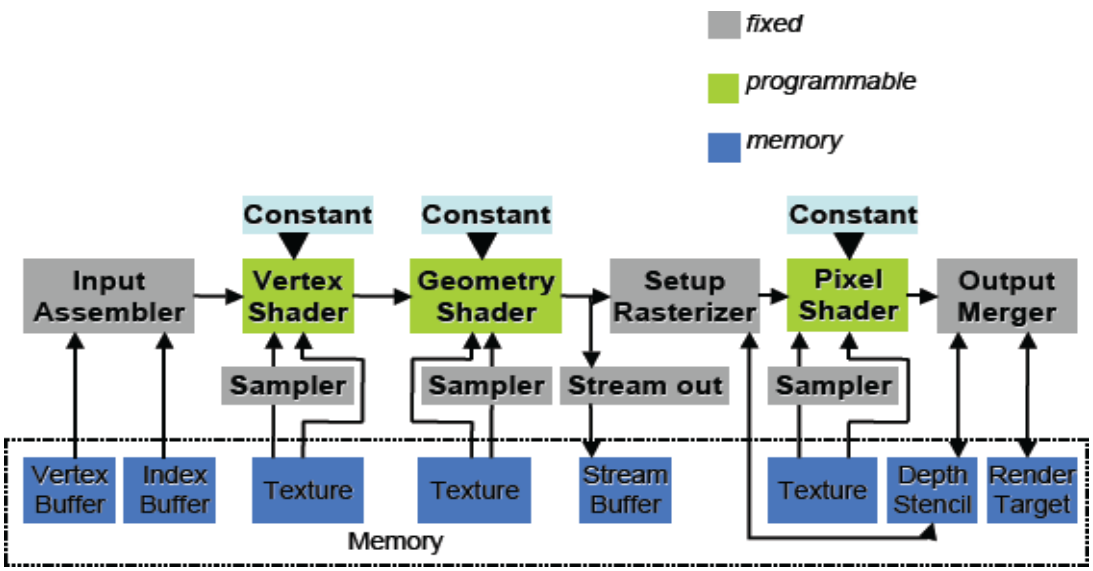
1.4. GPGPU Mimari Modeli

CUDA mimarisi ve platformu temel olarak tek buyrukta çok sayıda iş parçacığı (Single Instruction Multiple Threads) çalıştırma prensibine dayalıdır. OpenCL ile gerçeklenmiş işlemciler de temelde aynı mantığa dayalı olsa da bu şekilde isimlendirilmezler. Grafik işlemcilerin mimari yapısı Şekil 1.3'te gösterilmiştir. Normal işlemcilerde çok yer kaplayan ve karmaşık bir kontrol yapısı, büyük boyutlarda ve çok aşamalı önbellekler bulunurken, grafik işlemcilerde kontrol ve önbellek az yer kaplar ve basittir. Kalan silikon alanı ise işlem birimleri ile doldurulmuş olup, paralel işlem kapasitesi maksimize edilmiştir.

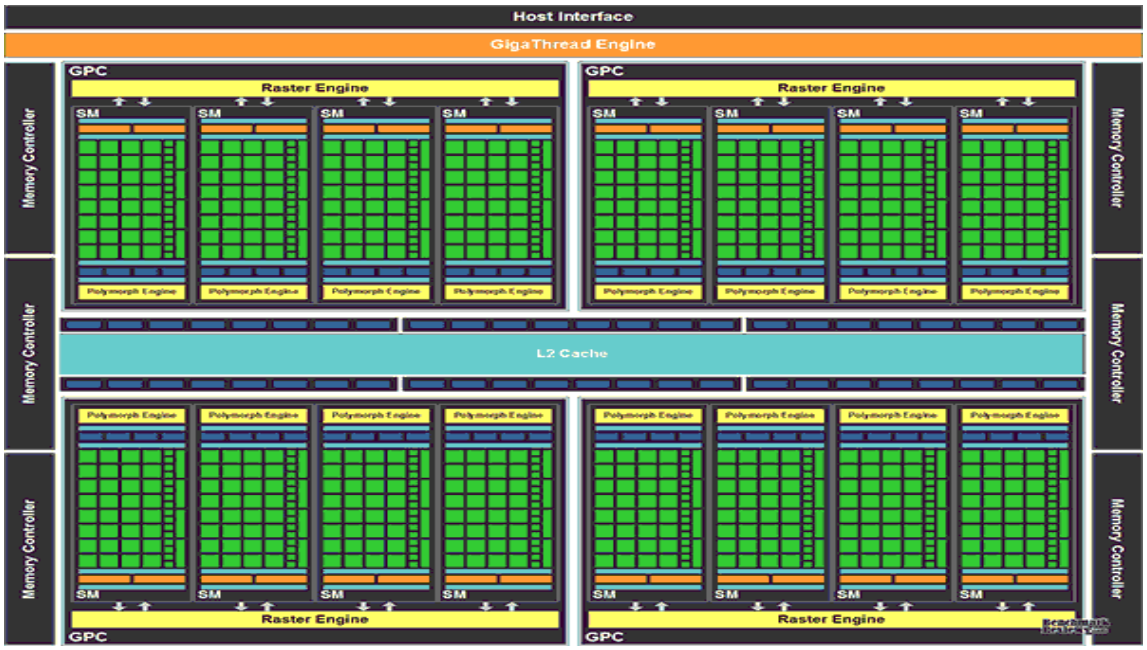
Şekil1. 5 CPU ve GPU mimarileri arasındaki farklar

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 1.6'da 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.



Şekil1. 6 DirectX 10 programlanabilir işlem hattı mimarisi



Şekil1. 7 Genel amaçlı grafik işlemcilerin iç yapısı

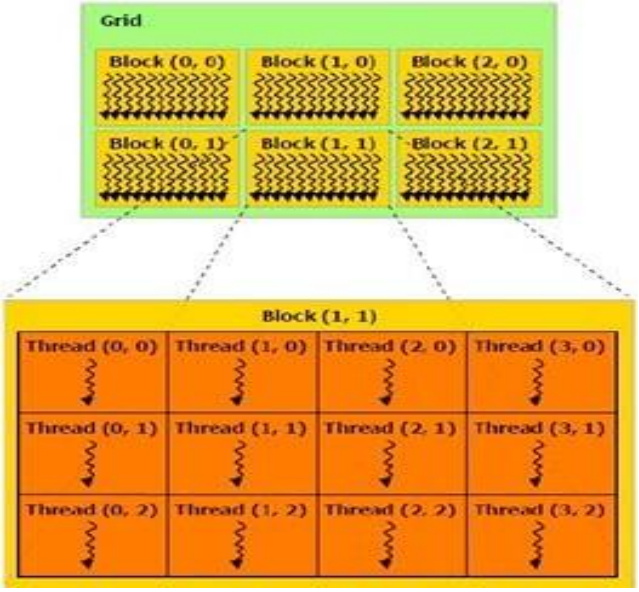
Şekil 1.7'de görülen grafik işlemcisi mimarisindeki akış çoklu-işlemcisi (SM), genel amaçlı işlemci mimarisindeki çekirdek kavramına benzemektedir. CPU'lardaki çekirdekler birden çok boru hattı ve çok sayıda işlem birimi içerebildiği gibi, SM'ler de çok sayıda akış işlemcisi (SP) içerir. Akış işlemcileri tek buyruk çok veri (SIMD) prensibi ile işlem yapabilme kapasitesine sahiptir. Tek bir buyruktaki işlemi aynı saat vuruşunda tasarımdan tasarıma değişmekle birlikte 2, 3, 4, 8 veya 16 kelimelik veri üzerine uygulayabilmektedir. GPGPU üzerinde yapılan işlemlerin çoğu basit aritmetik mantık işlemleri şeklindedir. Daha karmaşık ve donanımsal olarak gerçeklenmiş olan trigonometrik, logaritmik vb. fonksiyonlar için özelleşmiş işlem birimleri mevcuttur.

1.5. 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 1.8'de modern GPGPU dillerinden CUDA programlama diline ait programlama modeli yapısı, Şekil 1.9'da 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 1.8'de 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



Şekil1. 8 CUDA dili programlama modeli yapısı

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);
}
```

Şekil1. 9 Matris toplama işleminin C'de ve CUDA'da gerçeklenmesi

Şekil 1.9'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.

1.6 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

1.6.1 C for Graphics

C for Graphics (Cg) , 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.

1.6.2 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.

1.6.3 BrookGPU

Stanford Üniversitesinde geliştirilmiş olan BrookGPU , 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.

1.6.4 CUDA

CUDA (Compute Unified Device Architecture) , 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.

1.6.5 Direct Compute

Direct Compute , 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. 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.

Heterojen platform ismi, CPU, GPU, FPGA ve DSP kartları gibi donanımsal ve işlevsel olarak birbirinden farklı olan cihazların iki ya da daha fazlasının bir arada bulunduğu sistemlere verilen isimdir. OpenCL heterojen platformlar üzerinde paylaşılan bir kodun çalıştırılmasını sağlayan C++ tabanlı bir dildir. OpenCL kullanarak heterojen bir platformda istenilen cihazlar üzerinde aynı kod koşturulabilir. Heterojen platform sunucu ve hedef cihaz ya da cihazların birbirine bir PCIe (Peripheral Component Interconnect Express) ara yüzü ile bağlanmasını ifade edebileceği gibi son zamanlarda oldukça yaygın kullanılan "chip içinde sistem" (SoC) adıyla bilinen tek entegre devre içinde CPU, FPGA ya da GPU nun bir arada bulunmasından da oluşabilir. OpenCL, Khronos grup olarak bilinen bir konsorsiyum tarafından ortaya konulmuştur. Apple, Intel, Qualcomm, Advanced Micro Device (AMD), Nvidia, Altera, Samsung, ARM holdings ve Vivante bu konsorsiyumun üyeleridir. Khronos konsorsiyumu grafik, medya ve paralel programlama standartlarını belirlemektedir. OpenCL destekleyen donanımsal platformlar oldukça geniş donanım ailelerini kapsamaktadır. Bu platformlara her geçen gün yeni ürünler eklenmektedir. Bu ürünlerin ortak özelliği çoklu çekirdek yapısına sahip olmalarıdır.

2.1 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

2.1.1 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.

2.1.2 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.

2.1.3 Çekirdek nesneleri

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

2.1.4 Programlar

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

2.1.5 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.

2.1.6 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.

2.1.7 Komut kuyrukları

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

2.1.8 Ev sahibi programlar

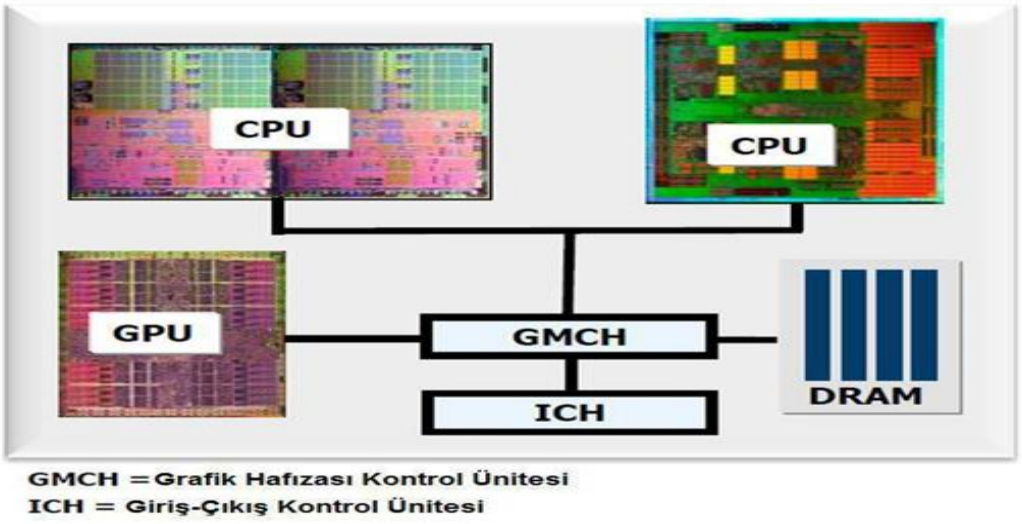
Ev sahibi program, çekirdek fonksiyonların hesaplama aygıtı üzerinde yürütülmesi için gerekli bağlamı hazırlayan ve yürütme işlemini düzenleyen programdır. Ev sahibi program CPU üzerinde çalışır fakat CPU aynı zamanda bir hesaplama aygıtı olarak da kullanılabilir. Çekirdek fonksiyonların aygıtlar üzerinde 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.

2.1.9 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.

2.2 OpenCL mimarileri

Heterojen platform bir ya da daha fazla CPU, GPU, DSP işlemcisi, FPGA ve hızlandırıcı bazı kartlardan oluşabilir. Heterojen sistemler, işlemlerin yalnızca CPU üzerinde yapıldığı sistemlerden birçok açıdan farklılık gösterir. Aslında bu farklılıklar heterojen sistemlerin neden ortaya çıktığının da cevabıdır. Bir sistemin gerçekleştireceği tüm işlemleri CPU üzerinde yapmak CPU'a fazla yük binmesine neden olur. CPU'ların temel çalışma frekansları her ne kadar diğer hızlandırıcı kartlardan fazla da olsa her işlemin CPU tarafından yapılması işlemlerin belli bir sıraya sokulmasını gerektirir ve dolayısıyla işlemleri yavaşlatır. İstenilen performansın sağlanması için sisteme fazladan CPU eklenirse hem sistemin maliyetini arttırır hem de güç harcaması oldukça yükselir. Bu sorunların çözümü içim daha az güç harcayan ve daha ucuza mal olabilecek kartlar heterojen sisteme dâhil edilerek sistemin maliyeti azaltılabilir. Aynı zamanda GPU, FPGA ve diğer hızlandırıcı kartlar daha az güç tükettiği için de sisteme avantaj sağlar. İşlem performansı açısından da bu cihazlar kendilerine has mimarileri sayesinde paralel hesaplama yapabilecek kabiliyetlere sahiptir. Dolayısıyla heterojen bir sistem kullanmak hem güç tüketimini azaltmak, hem performansı arttırmak hem de sistemi daha ucuza mal etmek bakımından avantajlıdır. Bir heterojen sistem örneği Şekil 1.10'da görülebilir.



Şekil1. 10 Bir heterojen platforma örneği

Şekil 1.10‘da gösterilen örnek heterojen sistemde bir veri yolu üzerinden iki CPU birbirine bağlanmıştır. Her iki CPU’nun kontrol ettiği ve yürüttüğü işlemler mevcuttur. Grafik hafızası kontrol birimi ise GPU’nun RAM bellekten veri okumasını ya da yazmasını sağlar. Giriş/çıkış kontrol ünitesi ise dışarıdan bilgi almaya ya da dışarıya bilgi çıktısı üretmeye kanal sağlar. Benzer şekilde birçok heterojen sistem mimarisi kurmak mümkündür. Mevcut olan ihtiyacı karşılayacak şekilde heterojen sistemlerin tasarımı yapılır. Günümüzde heterojen sistem mimarisi oldukça yaygın olarak kullanılmaktadır. Cep telefonları heterojen sistemlere örnek olarak gösterilebilir. Cep telefonlarında kullanılan CPU’lar hızlı veri işlemlerinin gerçekleştirilmesini, cep telefonu işletim sistemi üzerindeki çalışacak uygulamaların koşturulmasını sağlarken,sisteme gömülü olarak entegre olan bir DSP üzerinden de dışarıdan alınan ses verisi sayısallaştırılarak bit verileri haline dönüştürülür. Daha sonra bu bit verileri tekrar kulağın duyabileceği ses sinyallerine çevrilir. Böylece DSP, CPU’dan bağımsız olarak kendi görevini yerine getirir. Aynı zamanda bunu düşük güç tüketimiyle yaptığı için batarya ömrünün de uzamasını sağlamış olur

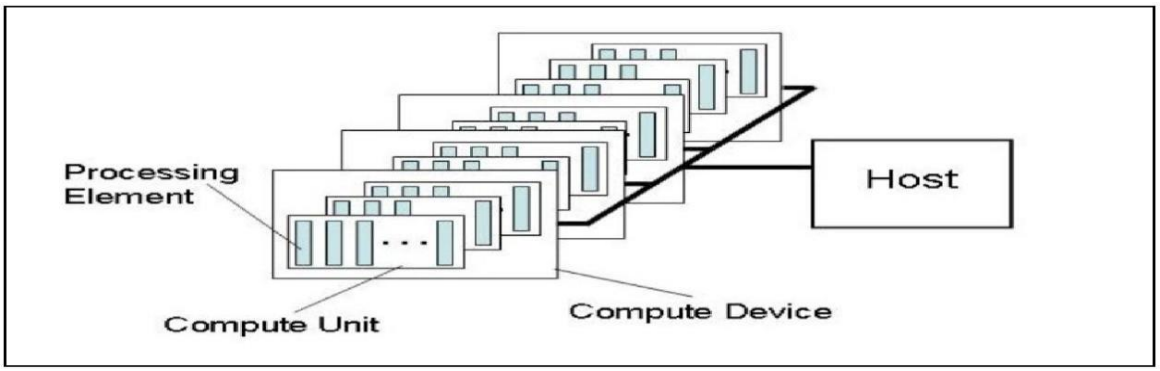
2.3 OpenCL Çalışma Modeli

OpenCL çalışma modeli aşağıdaki modellerin birleşimi şeklinde tanımlanır:

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

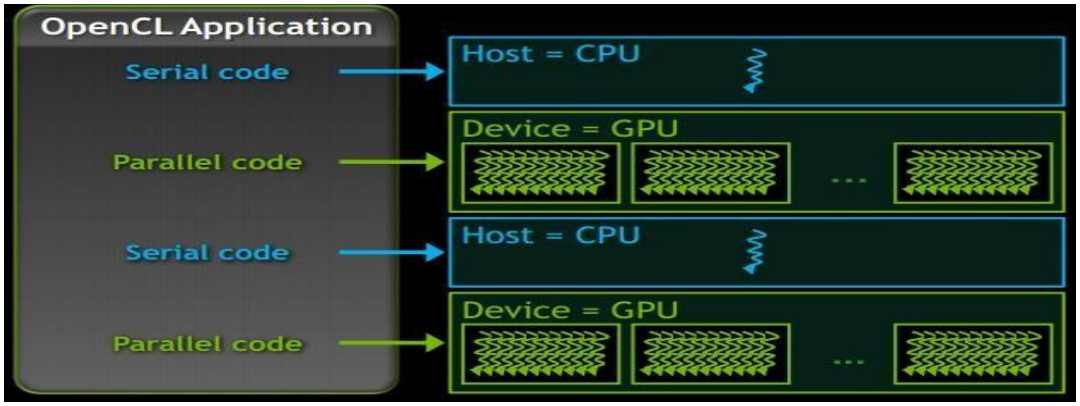
2.3.1 OpenCL platform modeli

Genel amaçlı işlemler için kullanılabilen grafik işlemcileri yalnız başına çalışamamaktadır. Her zaman bir platformda genel amaçlı işlemci veya işlemcilerin yanında, bir ek işlemci olarak yer almak zorundadır. Gelişen teknoloji ile birlikte aynı yonga üzerinde hem grafik işlemci hem de genel amaçlı işlemciler bulunabilmektedir. Her halükarda, işletim sisteminin üzerinde çalışması gereken bir genel amaçlı işlemci, grafik işlemcisinin de bu CPU ile haberleşmesini sağlayan bir sürücü yazılımı bulunmalıdır.
Önceden bahsedildiği gibi OpenCL yalnızca grafik işlemcileri değil, sinyal işleyici veya FPGA gibi cihazları da desteklemektedir. OpenCL destekli cihazların her birine OpenCL modelinde işlem aygıtı (compute device) denilmektedir. İşlem aygıtlarının hepsi sunucu (host) adı verilen genel amaçlı işlemciye bağlıdır. OpenCL de CUDA da aslında çalışan programın yalnızca bir bölümünü oluşturmaktadır. İşletim sistemi ile ilgili çağrıları, verileri okuma-yazma ve giriş çıkış işlemlerinin hepsi sunucu kısmında yürütülmektedir. OpenCL'in çalışma modeli ile ilgili ayrıntılı bilgiler bölüm 2.3.2'te verilecektir. Şekil 1.11'de OpenCL'in platform modeli gösterilmiştir. İşlem aygıtları bir sunucuya bağlıdır. İşlem aygıtları işlem birimlerine ayrılmaktadır. İşlem birimlerinin mimari yapıdaki karşılığı akış çoklu-işlemcisidir. Aygıtlarda bir veya birden fazla işlem birimi olabilir. İşlem birimi de işlem elementlerine bölünmektedir. İşlem elementlerinin mimarideki karşılığı da akış işlemcisidir.



Şekil1. 11 OpenCL platform modeli

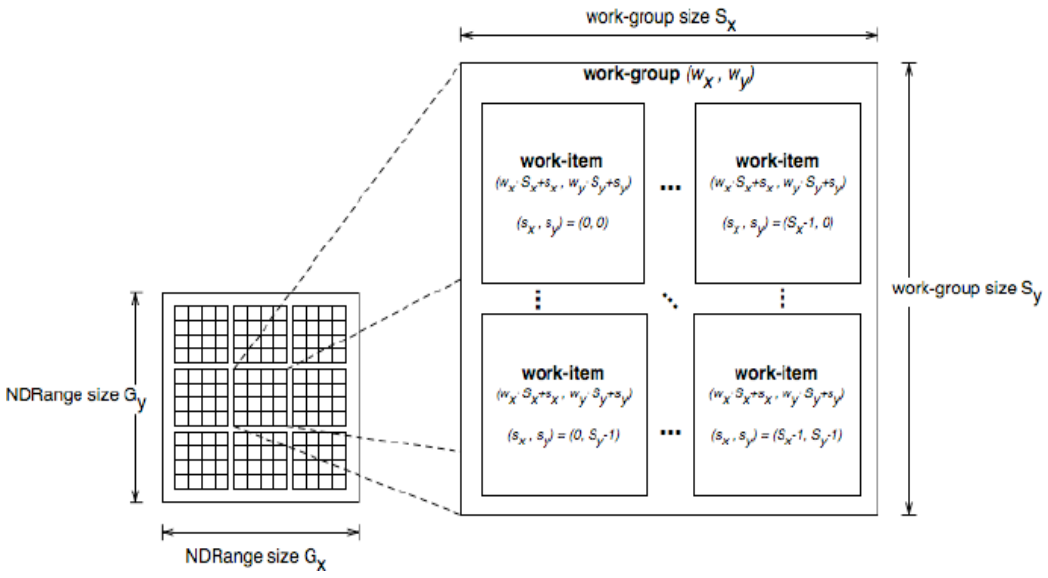
GPGPU platformlarında sunucu ve işlem aygıtlarının çalışma biçimi şekil 1.12 'de gösterilmiştir. Görüldüğü gibi genel amaçlı işlemci aslında grafik işlemcisine paralel hale getirilmiş iş parçacıkları gönderip sonuçları tekrar kendisinde toplamaktadır.



Şekil1. 12 GPGPU paralelleştirmesinin çalışma biçimi

2.3.2 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 1.13 NDRange endeks alanını, iş gruplarını ve iş öğelerini göstermektedir.



Şekil1. 13 NDRange endeks alanı, iş grupları ve iş öğeleri

2.3.3 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:

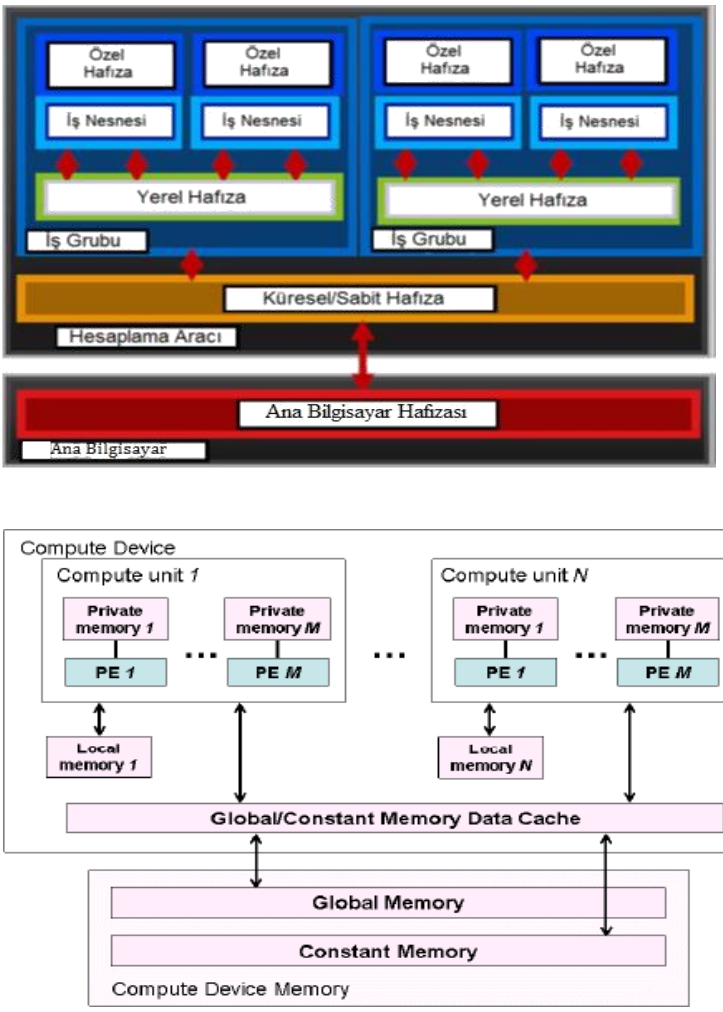
- Genel bellek (global memory): Tüm iş öğelerinin okuma ve yazma için erişimine açık olan bellek bölgesidir.

- 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 1.14'deOpenCL bellek modeline göre hesaplama aygıtı içerisindeki bellek bölgesi çeşitleri gösterilmiştir.



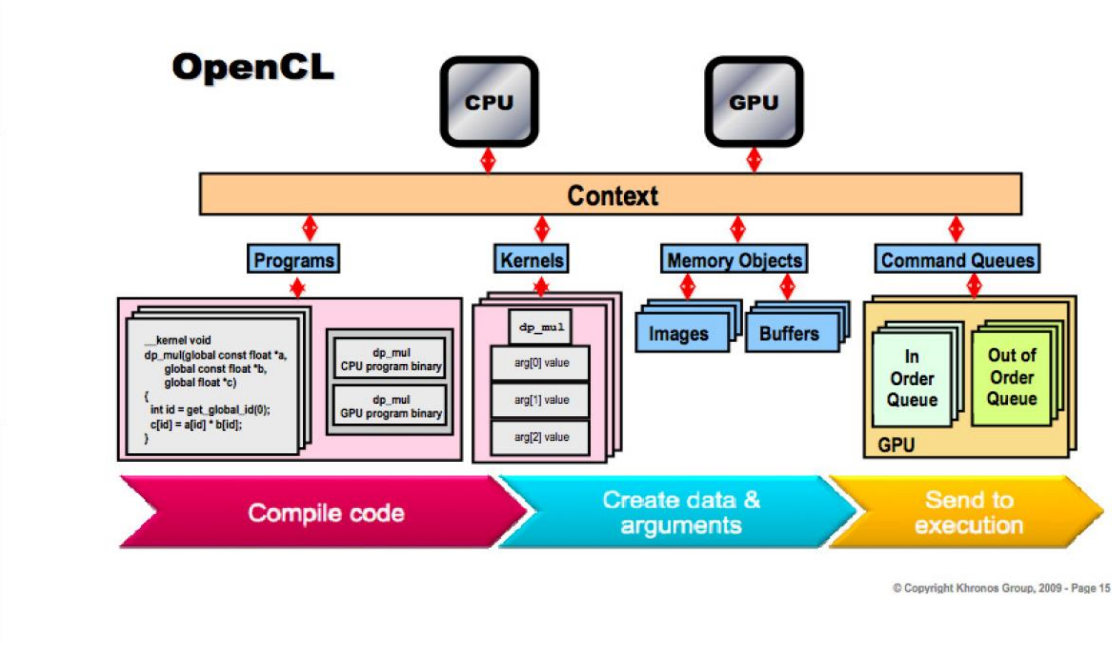
Şekil1. 14 Bellek bölgesi çeşitleri

2.3.4 OpenCL Programlama Modeli

OpenCL, veri paralel ve görev paralel programlama modellerini destekler. Veri paralel programlama modelinde aynı çekirdek fonksiyonu 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 oluşturulur ve bu fonksiyonlar farklı verilerle aynı anda paralel çalışan iş parçacıkları şeklinde yürütülür (Banger, 2013).

2.4 OpenCL Programlama

OpenCL'in data yı paralel işleyecek kısmı yani kernel kodunun C diliyle yazılması gerekmektedir.OpenCL programcıya yazdığı kodun her ortamda çalışabildiği bir platform sunar.(CPU's,GPU's,DSPs…)



Şekil1. 15 OpenCL Programlama Şeması

- **OpenCL Data Tipleri**

OpenCL C data tipleri : Scalar data,vector data,Abstract data,Reserved data,Other data types.

Table 4.1. OpenCL scalar data types (required minimum)

Scalar data type	Purpose
bool	A Boolean condition: true (1) or false (0)
char	Signed two's complement 8-bit integer
unsigned char/uchar	Unsigned two's complement 8-bit integer
short	Signed two's complement 16-bit integer
unsigned short/ushort	Unsigned two's complement 16-bit integer
int	Signed two's complement 32-bit integer
unsigned int/uint	Unsigned two's complement 32-bit integer
long	Signed two's complement 64-bit integer
unsigned long/ulong	Unsigned two's complement 64-bit integer
half	16-bit floating-point value, IEEE-754-2008 conformant
float	32-bit floating-point value, IEEE-754 conformant
intptr_t	Signed integer to which a void pointer can be converted
uintptr_t	Unsigned integer to which a void pointer can be converted
ptrdiff_t	Signed integer produced by pointer subtraction
size_t	Unsigned integer produced by the size of operator
void	Untyped data

Şekil1. 16 Scalar Data Tipleri

Table 4.3. OpenCL vector data types

Vector data type	Purpose
char*n*	Vector containing n 8-bit signed two's complement integers
uchar*n*	Vector containing n 8-bit unsigned two's complement integers
short*n*	Vector containing n 16-bit signed two's complement integers
ushort*n*	Vector containing n 16-bit unsigned two's complement integers
int*n*	Vector containing n 32-bit signed two's complement integers
uint*n*	Vector containing n 32-bit unsigned two's complement integers
long*n*	Vector containing n 64-bit signed two's complement integers
ulong*n*	Vector containing n 64-bit unsigned two's complement integers
float*n*	Vector containing n 32-bit single-precision floating-point values

Şekil1. 17 Vectör Data Tipleri

- **OpenCL Programlama Adımları**

Host: OpenCL programını çalıştırılacağı CPU ve ana belleğin bulunduğu PC.
Device: Host'a bağlı ve host tarafından kontrol edilen FPGA, ekran kartı gibi cihazlardır.

OpenCL Devices

cl_device_type	Description
CL_DEVICE_TYPE_CPU	OpenCL device that is the host processor.
CL_DEVICE_TYPE_GPU	OpenCL device that is a GPU.
CL_DEVICE_TYPE_ACCELERATOR	OpenCL accelerator (e.g., IBM Cell Broadband).
CL_DEVICE_TYPE_DEFAULT	Default device.
CL_DEVICE_TYPE_ALL	All OpenCL devices associated with the corresponding platform.

Şekil1. 18 OpenCL Devices Tablosu

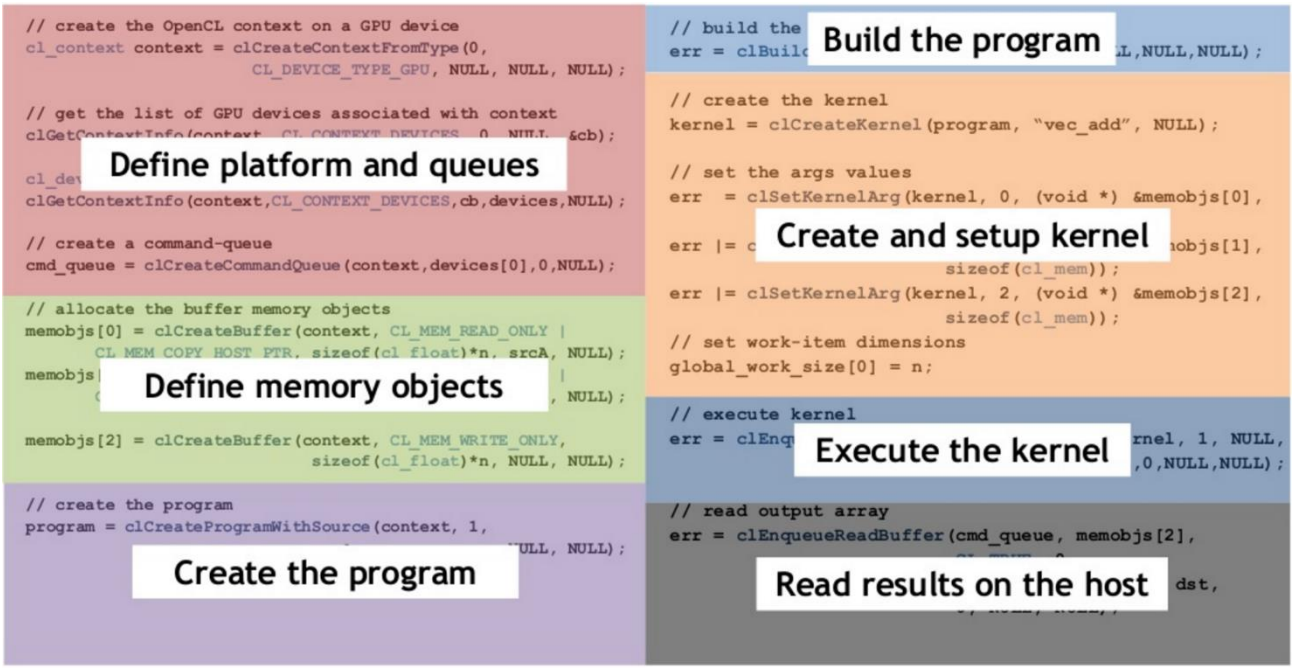
```
cl_int clGetDeviceIDs( cl_platform_id platform,
                       cl_device_type device_type,
                       cl_uint num_entries,
                       cl_device_id *devices,
                       cl_uint *num_devices );
```

Çekirdek Fonksiyon (Kernel): Cihaz üzerinde çalıştırılabilen fonksiyonlara verilen isimdir.

```
cl_kernel clCreateKernel( cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret );
```

Programlama Adımları

- OpenCL aygıtları taranır: Sistemde OpenCL'i destekleyen cihazlar var mı tespit edilir.
- Kontext: OpenCL uygulamasının dış çerçevesi gibi düşünebiliriz.
- Programlar oluşturulur: Kontext içinde bir veya daha fazla OpenCL cihazını kullanmak için,
- Program içinden çekirdek fonksiyonları belirlenir.
- Memory nesneleri ile bellekte gerekli yer ayırma ve kopyalama işlemlerini modellenir.
- Gerekli veri aktarımları yapılır,
- Çekirdek fonksiyonları "komut kuyruğuna" yerleştir
- Programı çalıştırılır.



Şekil1. 19 OpenCL Programlama Adımları

- **OpenCL Programlama Örnekleri**
- **Array Toplama İşlemi**

OpenCL öğrenenlerin en temel gerçeklediği program iki dizgenin toplamını veren programdır. Bu program, temel kavramları ve ilişkilerini güzelce açıklar.

Kod

Platform'dan Device seçimi yapılıp Context oluşturulur ve Kaynak derlenir. Hem RAM'de hem de cihaz belleğinde işleçleri ve sonucu tutacak Buffer'lar oluşturulur. Aradaki veri aktarımı ve komutların çalıştırılması için CommandQueue oluşturulur. Kernel'e argümanlar paslanır ve sonuçları RAM'e geri kopyalanır. Sonuçlar ekrana yazdırılır. (İlgili açıklamaları kodun içindeki yorum satırlarında da bulabilirsiniz.)

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <random>
#include "CL/cl2.hpp"
int main()
{
        // OpenCL Platformu: Intel için Intel OpenCL
        std::vector<cl::Platform> platforms;

        cl::Platform::get(&platforms); // Statik Get Platform metodu
        for (auto var : platforms)
        {
                std::cout << var.getInfo<CL_PLATFORM_NAME>() << std::endl;
                // Her Platform için platform ismini ekrana yazdır
        }

        // Kodu çalıştırdığım bilgisayarda bu platforma ait 2 cihaz var
        // Biri normal CPU diğeri ise Intel entegre grafik kartı
        auto selected_platform = platforms[0];

        std::vector<cl::Device> devices;
        selected_platform.getDevices(CL_DEVICE_TYPE_ALL, &devices);
        for (int i = 0; i < devices.size();i++)
        {
                std::cout << "[" << i <<"] "<< devices[i].getInfo<CL_DEVICE_NAME>() << std::endl;
                // Benzer sekilde cihazlari sirala
        }

        // Kullanıcıdan cihaz seçmesini isteyelim.
        int device_n = 0;
        std::cout << "Select device no: ";
        std::cin >> device_n;
        if (device_n > devices.size() || device_n < 0)
        {
                return 255;
        }
        auto default_device = devices[device_n];

        //  Bir çalışma contexti oluştur.
```

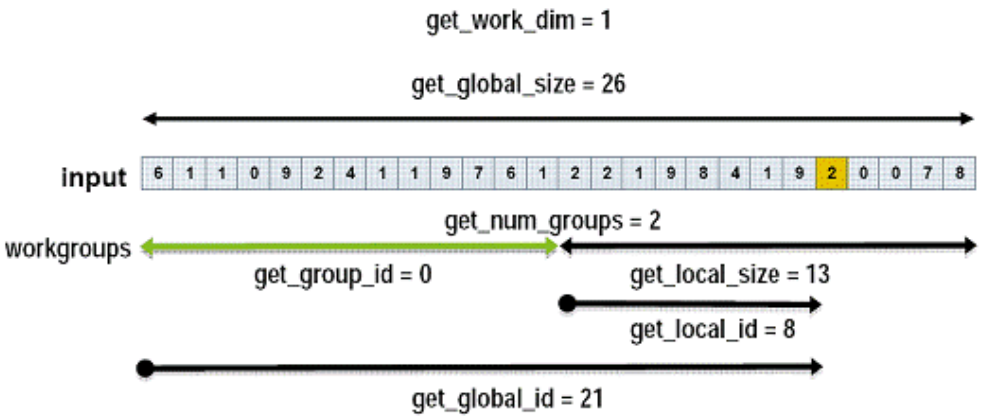
```
41          cl::Context context(default_device);
42
43          // Yazdığımız programı oluşturan GPU kodlarını tutan vector
44          cl::Program::Sources sources;
45
46          std::string code = ""
47                  "void kernel simple_add(global const int* A, global const int* B, global int* C)"
48                  "{"
49                  "size_t row = get_global_id(0);"
50                  "C[row] = A[row] + B[row];"
51                  "}";
52          sources.push_back(code);
53          cl::Program program(context, sources);
54          // Bu context ve kaynak kodlarıyla programı oluştur
55          // Derle!
56          if (!program.build({ default_device }) == CL_SUCCESS)
57          {
58                  std::cerr << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device) << std::endl;
59                  // Derlenmediyse hata mesajlarini yazdir
60                  return 255;
61          }
62
63          // GPU Buffer oluştur
64          size_t size = 16;
65          size_t bufferSize = sizeof(int) * size;
66          cl::Buffer bufferA(context, CL_MEM_READ_WRITE, bufferSize);
67          cl::Buffer bufferB(context, CL_MEM_READ_WRITE, bufferSize);
68          cl::Buffer bufferC(context, CL_MEM_READ_WRITE, bufferSize);
69
70          // Bellek ile GPU arasında git gel yapacak bir iş kuyruğu oluştur
71          cl::CommandQueue queue(context, default_device);
72
73          // A B dizgeleri
74          int *A = new int[size];
75          int *B = new int[size];
76          int *C = new int[size];
77          std::fill_n(A, size, 1);
78          std::fill_n(B, size, 2);
79          // Ekrana yazdır
```

```
80          for (int i = 0; i < size; i++)
81          {
82                  std::cout << A[i] << " ";
83          }
84          std::cout << std::endl;
85
86          // Ekrana yazdır
87          for (int i = 0; i < size; i++)
88          {
89                  std::cout << B[i] << " ";
90          }
91          std::cout << std::endl;
92
93          // GPU'ya aktar
94          queue.enqueueWriteBuffer(bufferA, CL_TRUE, 0, bufferSize, A);
95          queue.enqueueWriteBuffer(bufferB, CL_TRUE, 0, bufferSize, B);
96          // Programımızdaki simple_add kerneli
97          cl::Kernel kernel(program, "simple_add");
98
99          // Kernel parametreleri
100         kernel.setArg(0, bufferA);
101         kernel.setArg(1, bufferB);
102         kernel.setArg(2, bufferC);
103         // 16 work item kullanarak bu kerneli isle
104         queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(size));
105
106         // Sonuçları belleğe oku
107         queue.enqueueReadBuffer(bufferC, CL_TRUE, 0, bufferSize, C);
108
109         // Ekrana yazdır
110         for (int i = 0; i<size;i++)
111         {
112                 std::cout << C[i] << " ";
113         }
114         std::cout << std::endl;
115 }
```

ÇIKTI

```
Intel(R) OpenCL
[0] Intel(R) UHD Graphics 620
[1] Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
Select device no: 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

- **OpenCL Index Tanımları**



Tek boyutlu 16 adet global work-item oluşturduk. Her bir dizge elemanına work-item numarası(get_global_id) ile erişebiliriz.

get_global_id = get_group_id*get_local_size+get_local_id

Eksik work-item sayısı(12) girildiğinde aşağıdaki gibi bir çıktı oluşabilir:

```
Intel(R) OpenCL
[0] Intel(R) UHD Graphics 620
[1] Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
Select device no: 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0
```

- **Matris Çarpımı**

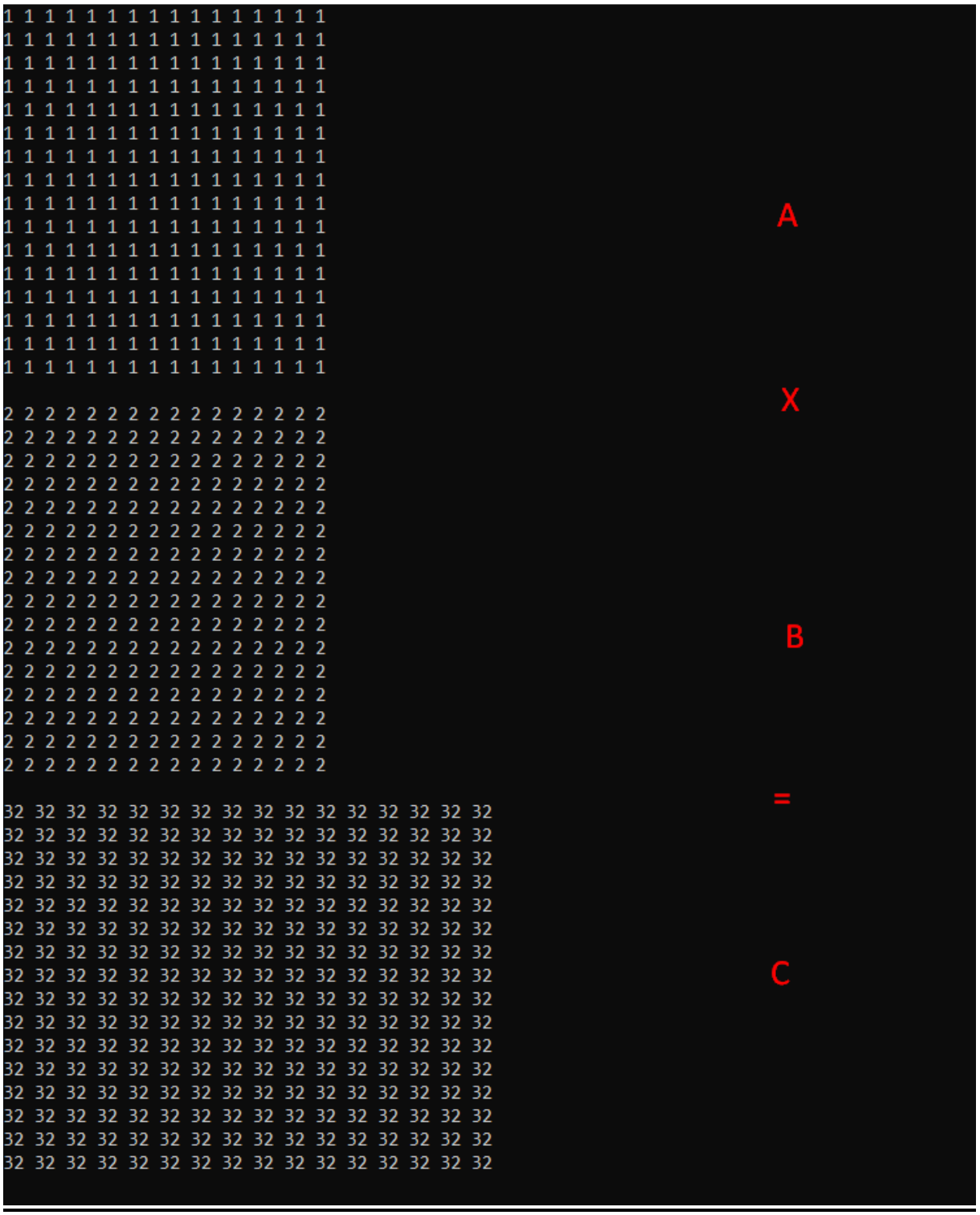
  En çok paralelleştirilen operasyonlardan birisi matris çarpımıdır. En hızlı matris çarpım algoritmasının bile karmaşıklığı yüksek olduğu için halihazırda bağımsız parçalar halinde yürütülmeye müsait olan bu işlem paralel programlama için önem arz eder. Matris çarpımı, 3D Rendering ve makine öğrenmesi gibi GPU'ların tercih edildiği uygulamalar için vazgeçilmezdir.

```
void kernel matmul(const int K, const int M, const int N, global const int* A, global const int* B, global int* C)
{
        int global_row = get_global_id(0);
        int global_col = get_global_id(1);
        int sum = 0;
        for(int i = 0; i<M;i++){
                sum += A[global_row*M+i] * B[i*K+global_col];
        }
        C[global_row*K+global_col] = sum;
};
```

```
// Onceki boilerplate kod
// GPU Buffer oluştur
size_t size = 16; // Butun matrisler 16x16
size_t size_orig = size; // 16
size *= size; // 16*16
size_t bufferSize = sizeof(int) * size;
cl::Buffer bufferA(context, CL_MEM_READ_WRITE, bufferSize);
cl::Buffer bufferB(context, CL_MEM_READ_WRITE, bufferSize);
cl::Buffer bufferC(context, CL_MEM_READ_WRITE, bufferSize);

// Bellek ile GPU arasında git gel yapacak bir iş kuyruğu oluştur
cl::CommandQueue queue(context, default_device);

// A B dizgeleri
int *A = new int[size]; // 16*16
int *B = new int[size]; // 16*16
int *C = new int[size]; // 16*16
std::fill_n(A, size, 1);
std::fill_n(B, size, 2);
// Ekrana yazdır
for (int i = 0; i < size; i++)
{
        std::cout << A[i] << " ";
        if ((i + 1) % size_orig == 0)
        {
                std::cout << std::endl;
        }
}
std::cout << std::endl;

// Ekrana yazdır
for (int i = 0; i < size; i++)
{
        std::cout << B[i] << " ";
        if ((i + 1) % size_orig == 0)
        {
                std::cout << std::endl;
        }
}
std::cout << std::endl;
```

```

// GPU'ya aktar
queue.enqueueWriteBuffer(bufferA, CL_TRUE, 0, bufferSize, A);
queue.enqueueWriteBuffer(bufferB, CL_TRUE, 0, bufferSize, B);
// Programımızdaki matmul kerneli
cl::Kernel kernel(program, "matmul");

// Kernel parametreleri
kernel.setArg(0, size_orig); // K
kernel.setArg(1, size_orig); // M
kernel.setArg(2, size_orig); // N
kernel.setArg(3, bufferA);
kernel.setArg(4, bufferB);
kernel.setArg(5, bufferC);
// 16*16 work item kullanarak bu kerneli isle
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(size_orig, size_orig)); // 16*16 work-item

// Sonuçları belleğe oku
queue.enqueueReadBuffer(bufferC, CL_TRUE, 0, bufferSize, C);

// Ekrana yazdır
for (int i = 0; i<size;i++)
{
        std::cout << C[i] << " ";
        if ((i + 1) % size_orig == 0)
        {
                std::cout << std::endl;
        }
}
std::cout << std::endl;
```

ÇIKTI



```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
32  32  32  32  32  32  32  32  32  32  32  32  32  32  32  32
```

- **Vektör Addition**

- Kernel Code

```
__kernel void vector_add(__global const int *A, __global const int *B, __global   int *C)
{

    // Get the index of the current element to be processed
      int i = get_global_id(0);

    // Do the operation
     C[i] = A[i] + B[i];
}
```

Host Code

```
int main(void)
{
  // 2 tane Input vectoru yaratıyoruz.
  int i;
  const int LIST_SIZE = 10;
  float *A = (float*)malloc(sizeof(int)*LIST_SIZE);
  float *B = (float*)malloc(sizeof(int)*LIST_SIZE);

   for(i = 0; i < LIST_SIZE; i++)
     {
        A[i] = i;
        B[i] = i+5;
      }
// Kernel source kodunu source_str nın ıcıne gomuyoruz.

   FILE *fp;
  char *source_str;
  size_t source_size;
```

```
// Platform ve cihaz bılgıleri alınır.

cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
      &device_id, &ret_num_devices);


// OpenCL icerikleri yaratılır.
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

// Komut orgusu olusturulur.
cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

// Device uzerınde her deger ıcın ara bellekler olsturulur.
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
      LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
      LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      LIST_SIZE * sizeof(int), NULL, &ret);
```

```
// A ve B ınput vectorlerımızın degerleı GPU uzerındekı olusturulan ara belleklere yazılır
  ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
        LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
  ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
        LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

  // Kernel kaynagından program olusturulur.
  cl_program program = clCreateProgramWithSource(context, 1,
        (const char **)&source_str, (const size_t *)&source_size, &ret);

  // Program build edılır.
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);



   // Toplama yapacak OpenCL kernelı yaratılır.
   cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

   // Kernel kaynagındakı argumanlar tanıtılır.
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);



      // OpenCL kernelı execute edılır.
       ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
            &global_item_size, &local_item_size, 0, NULL, NULL);

       // GPU uzerındekı ara bellekte bulunan  sonuc degerleı Local varıable olan C ye yazılır.
       float *C = (float*)malloc(sizeof(int)*LIST_SIZE);
       ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), C, 0, NULL, NULL);


      // sonucları goruntulenır.
       for(i = 0; i < LIST_SIZE; i++)
          printf("%f + %f = %f\n", A[i], B[i], C[i]);

       // Hafızada yer tutan bolumler bosaltılır..
       ret = clFlush(command_queue);
       ret = clFinish(command_queue);
       ret = clReleaseKernel(kernel);
       ret = clReleaseProgram(program);
       ret = clReleaseMemObject(a_mem_obj);
       ret = clReleaseMemObject(b_mem_obj);
       ret = clReleaseMemObject(c_mem_obj);                    free(A);
       ret = clReleaseCommandQueue(command_queue);             free(B);
       ret = clReleaseContext(context);                        free(C);
                                                               return 0;
                                                             }
```

- **OpenCL' de Hello Word**

List 3.3: Hello World - kernel (hello.cl)

```
__kernel void hello(__global char* string)
{
string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = ',';
string[6] = ' ';
string[7] = 'W';
string[8] = 'o';
string[9] = 'r';
string[10] = 'l';
string[11] = 'd';
string[12] = '!';
string[13] = '\0';
}
```

List 3.4: Hello World - host (hello.c)

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_mem memobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;
cl_platform_id platform_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;

char string[MEM_SIZE];

FILE *fp;
char fileName[] = "./hello.cl";
char *source_str;
size_t source_size;
```

```
32
33  /* Load the source code containing the kernel*/
34  fp = fopen(fileName, "r");
35  if (!fp) {
36  fprintf(stderr, "Failed to load kernel.\n");
37  exit(1);
38  }
39  source_str = (char*)malloc(MAX_SOURCE_SIZE);
40  source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
41  fclose(fp);
42
43  /* Get Platform and Device Info */
44  ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
45  ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_devices);
46
47  /* Create OpenCL context */
48  context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
49
50  /* Create Command Queue */
51  command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
52
53  /* Create Memory Buffer */
54  memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL, &ret);
55
56  /* Create Kernel Program from the source */
57  program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
58  (const size_t *)&source_size, &ret);
59
60  /* Build Kernel Program */
61  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
62
63  /* Create OpenCL Kernel */
64  kernel = clCreateKernel(program, "hello", &ret);
65
66  /* Set OpenCL Kernel Parameters */
67  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
68
69  /* Execute OpenCL Kernel */
70  ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
71
72  /* Copy results from the memory buffer */
73  ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
74  MEM_SIZE * sizeof(char),string, 0, NULL, NULL);
75
76  /* Display Result */
77  puts(string);
78
79  /* Finalization */
80  ret = clFlush(command_queue);
81  ret = clFinish(command_queue);
82  ret = clReleaseKernel(kernel);
83  ret = clReleaseProgram(program);
84  ret = clReleaseMemObject(memobj);
85  ret = clReleaseCommandQueue(command_queue);
86  ret = clReleaseContext(context);
87
88  free(source_str);
89
90  return 0;
91  }
```

3. FPGA

FPGA, "Sahada Programlanabilir Kapı Dizileri" anlamına gelen "Field Programmable Gate Array" ifadesinin kısaltmasıdır. Kapı dizileri (gate array) ismi, FPGA'in yapısının Gate Array ASIC'lere benzemesinden dolayı verilmiştir. Sahada programlanabilir olması da FPGA'nın üretimden sonra programlanabilme özelliğine işaret eder (wordpress 2011).

FPGA'i kısaca, üretimden sonra istenen fonksiyona göre donanım yapısı kullanıcı tarafından değiştirilebilen tümleşik devre olarak tanımlayabiliriz. FPGA'i daha iyi anlayabilmek için, mantık kapılarının birbirinden bağımsız ve serbest olarak üretildiği ham bir tümleşik devre olarak hayal edebiliriz. Kullanıcının belirleyeceği tasarıma göre FPGA içindeki mantık kapıları birbirlerine bağlanabilir ve bu sayede istenen fonksiyonlar gerçekleştirilebilir. Yani teorik olarak sahip olduğu mantık kapısı kapasitesi dâhilinde akla gelen herhangi bir tümleşik devrenin yaptığı iş FPGA ile yapılabilir (instruments 2012) (Smith 2010).

Programlanabilir olması sayesinde FPGA'lar teknoloji dünyasında çok geniş bir yelpazede kullanılmaktadır. Ayrıca FPGA'ların gömülü işlemciler ve sayısal sinyal işleme blokları, PLL'lar, yüksek hızlı paralel ve seri giriş/çıkış desteği, harici bellek ara yüzleri, PCI Express ve Gigabit Ethernet gibi alıcı/ verici ara yüzleri gelişmiş saat ve güç dağıtım yönetimi gibi özellikleri birçok alanda FPGA'ların tercih edilmesini sağlamıştır. FPGA kullanımının bu denli popüler olmasının en önemli nedenlerinden birisi de, güçlü yazılım desteğidir. FPGA üreticilerinin sundukları gelişmiş PCB tasarım, modelleme ve benzetim yazılımları ile yapılandırılabilir hazır fikri mülkiyet çekirdekleri tasarımcıların işlerini oldukça kolaylaştırmaktadır. Bileşenlerin yerleşimlerini belirleme, aralarındaki bağlantıları tasarlama, zamansal analizler yapma ve tasarım değişikliklerini gerçekleştirme gibi işlemler FPGA tasarım akışında kolaylıkla yapılabilmektedir. FPGA üreticilerinin sundukları yazılımlar sayesinde tasarımlar sentezlenebilmekte ve FPGA cihazının içerisine otomatik olarak yerleştirilebilmektedir (Wilson 2015).

FPGA'nın tasarımcılara sunduğu en büyük avantaj paralel işlem yapabilme özelliğidir. Bu özelliği sayesinde çok daha hızlı işlem yapma olanağı sunmaktadır. Çok büyük ve karmaşık tasarımların FPGA üzerinde gerçekleştirilmesi hız, yer, zaman ve aynı zamanda güvenlik bakımından üstünlükler sunmaktadır. Ayrıca yeniden programlanabilme özelliği ile de tasarımcıya tasarımı güncelleyebilme olanağı sağlamaktadır. Deneme maliyeti düşerken zaman açısından da büyük avantajlar elde edilmektedir.

3.1.OpenCL ve FPGA

FPGA'lar doğal olarak paraleldir, bu nedenle OpenCL'in paralel bilgi işleme yetenekleri ile mükemmel bir uyum içindedir. FPGA, iş hattı paralelliği kabiliyetleri ile tipik veri veya görev paralelliğine bir alternatif sunar. Burada, görevler, ana bilgisayar programının etkileşimi olmaksızın önceki görevden farklı veriler kullanan, gönderme-alma yapılandırmasında oluşturulabilir. OpenCL, kodun tanıdık C programlama dilinde geliştirilmesine ve OpenCL tarafından sağlanan ek özelliklerin kullanılmasına olanak tanır. Çekirdek olarak adlandırılan bu uygulamalar FPGA tasarımcılarının düşük seviyeli HDL dillerini öğrenmek zorunluluğunu ortadan kaldırır ve kolaylıkla FPGA'e yüklenebilir.

Yazılım geliştiricileri ve sistem tasarımcıları için ve FPGA'da çalışacak kodları geliştirmede OpenCL'i kullanmanın çeşitli avantajları vardır:

Basitlik ve geliştirme kolaylığı: Çoğu yazılım geliştiricisi C programlama dilini bilir, ancak düşük seviyeli HDL dillerini bilenlerin sayısı azdır. OpenCL, yazılım geliştiricisini daha yüksek bir programlama seviyesinde tutarak sistemi daha fazla yazılım geliştiricisine açık hale getirir
Kod ayrımlaması: OpenCL kullanılarak, kod ayrımlanabilir ve performansa duyarlı kısımlar belirlenerek, FPGA'da çekirdekler olarak donanım hızlandırmasına tabi tutulabilir.

Performans: Watt başına performans, sistem tasarımının nihai hedefidir. FPGA kullanarak, yüksek performans ile enerji açısından verimli bir çözüm elde edilebilir.

Heterojen sistemler: OpenCL ile gerçek anlamda heterojen bir sistem tasarımı sağlamak için FPGA'ları, CPU'ları, GPU'ları ve DSP'leri sorunsuz bir şekilde hedefleyen çekirdekler geliştirebilir.

Kodun yeniden kullanımı: Yazılım geliştirmenin önemli kurallarından biri, kodun tekrar kullanılmasını sağlamaktır. Kodların tekrar kullanımı, genellikle yazılım geliştiricileri ve sistem tasarımcıları için zor bir hedeftir. OpenCL çekirdekleri, FPGA'ın farklı aileleri ve nesilleri için hedeflenebilecek taşınabilir kod geliştirilmesini ve kodun ömrünün uzun olmasını sağlar.

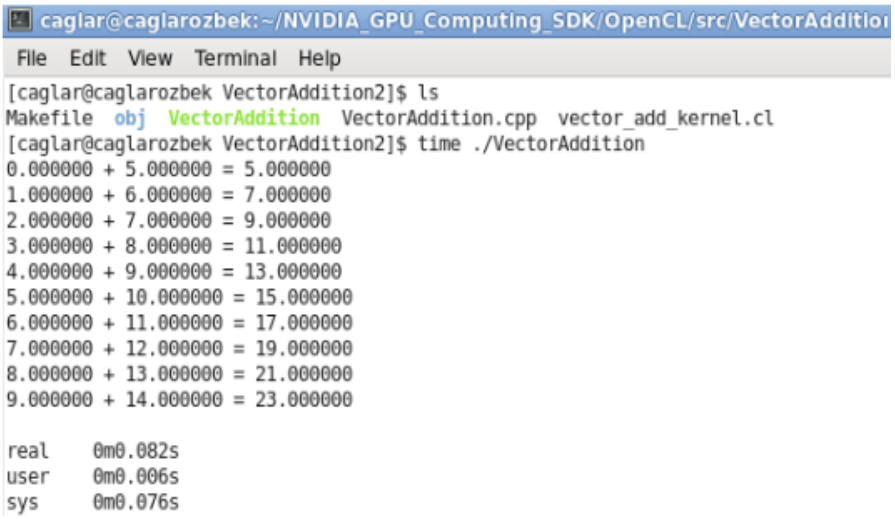
Çoğu FPGA üreticisi, FPGA'lar üzerinde OpenCL ile geliştirme için yazılım geliştirme araçları sağlar (Wiley ve Sons, 2014).

4.RAPORUN SONUCU

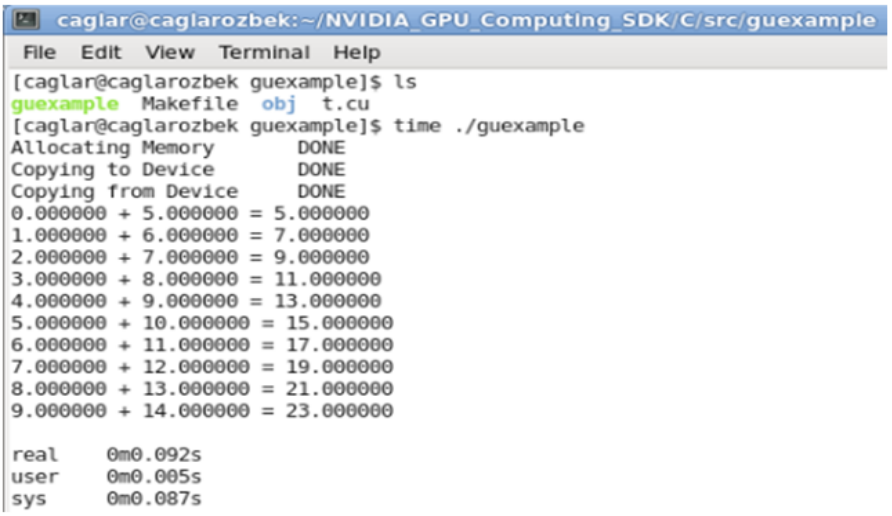
OpenCL; GPU, CPU ve diğer işlem birimlerinden oluşan heterojen ortamlarda GPGPU programlarını çalıştırmaya yarayan bir çatıdır. Sunmuş olduğu genel geçer API ile platform ve üretici bağımsızlığı sağlamaktadır. OpenCL standartlarının kabulünün gün geçtikçe artması ve önde gelen üreticilerin katkısı ile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

Öncelikle belirtmek gerekir ki GPU dillerinin performans arttırdığını görebilmek için ancak ağır işlem yükü olan kodlara optimize sekilde uygulanarak görülebilir.

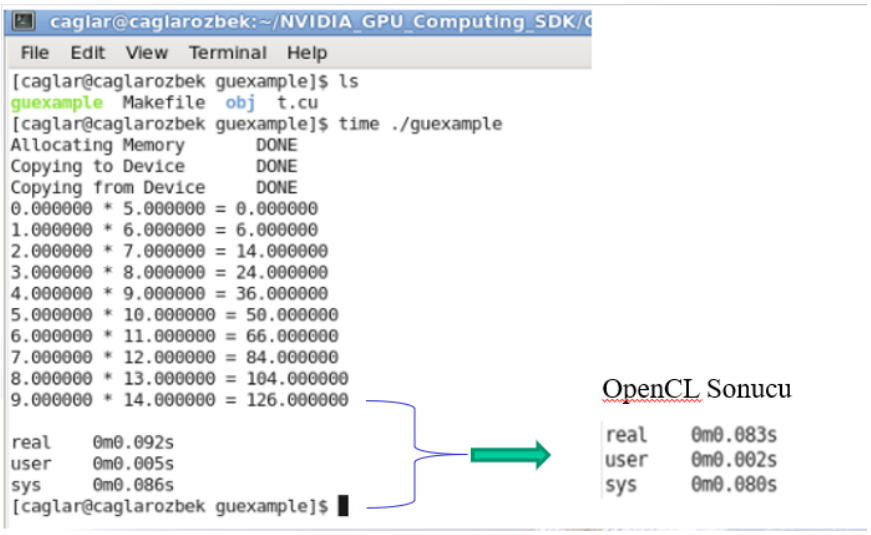
- Aşağıdaki yapılmış örneklerde OpenCL'in performans durumlarını görüyoruz.



OpenCL Vector Addition



Nvidia CUDA Vector Addition



Nvidia CUDA-OpenCL Vector Multiplication

- **Her iki örnekte de OpenCL ile %10 luk bir performans artışı sağlıyoruz.**

```
#include <stdio.h>
#include <iostream>
using namespace std;


static long double num_steps=8000000;
double step;
int main ()
{
  long double i;long double  x,pi,sum=0.0;
  step=1.0/(double)num_steps;
     for (i=0;i<num_steps; i++)
       {
          x=(i+0.5)*step;
           sum=sum+4.0/(1.0+x*x);
        }
   pi=step*sum;
   cout<< " pi = " << pi <<endl;
   return 0;
}
```

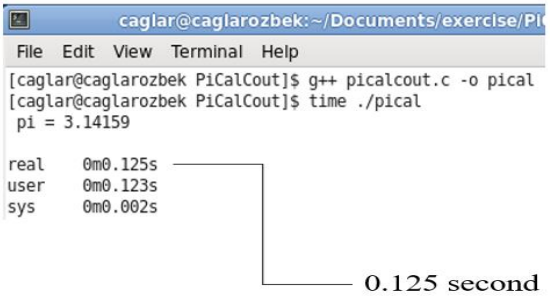
- Yandaki örnekte komutlar birbirine bağımlıdır.

- Loop içinde son değere kadar sonuçlar birbirini etkilemektedirler.

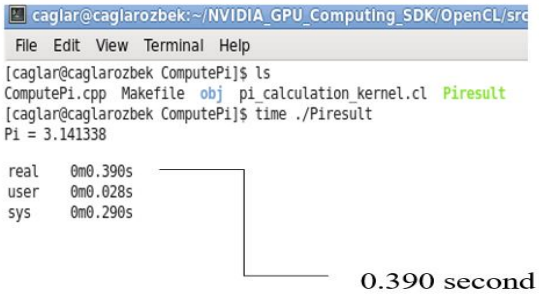
- Bu kod OpenCL'e uygulanmak için uygun durumda değildir.

- Öncelikle OpenCL'e uygulanacak kodun yapısı anlaşılmalıdır.Eğer doğru kod doğru şekilde OpenCL'e uygulanmaz ise performans düşer bu istenmeyen bir durumdur.

Simple C++ Code with 1M loop



Unoptimize OpenCL code



KAYNAKÇA

- **https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html**

- **Munshi, B. R. Gaster, T. G. Mattson, J. Fung ve D. Ginsburg, OpenCL programming guide, Pearson Education, 2011.**

- **R. J. Rost, OpenGL Shading Language, Addison-Wesley Professionnal, 2009.**

- **https://us.fixstars.com/products/opencl/book/OpenCLProgrammingBook/contents/**

- **https://cims.nyu.edu/~schlacht/OpenCLModel.pdf**

- **https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/**

- **OpenCL Programming Guide Version 4.2,Nvidia**

- **https://livebook.manning.com/book/opencl-in-action/table-of-contents/**

- **https://medium.com/@sddkal/opencl-9fca9c1ddc1f**