MULTI GPU

1. Giriş:

Modern hesaplama uygulamaları, tek bir GPU'nun işlem gücünü aşan büyük veri setlerini işlemeyi ve karmaşık işlemleri gerçekleştirmeyi gerektirebilir. Bu noktada Multi-GPU (çoklu GPU) kullanımı, birden fazla GPU'nun birlikte çalışarak işlemlerin süresini kısaltmalarını ve daha büyük problemleri çözmelerini mümkün kılar. CUDA, çoklu GPU'ları eşzamanlı ve verimli şekilde programlamaya olanak sağlar. Her bir GPU'nun kendi belleği ve işlemcileri vardır, ancak doğrudan veri aktarımı, senkronizasyon ve iş yükü paylaşımı CUDA API'ler ve yardımcı kütüphaneler (NCCL, Unified Memory gibi) aracılığıyla yönetilebilir. Multi-GPU özellikle Derin Öğrenme (Deep Learning), Bilimsel Hesaplamalar (Scientific Computing), Yüksek Başarımlı Hesaplama (HPC / Supercomputing), Görüntü ve Video İşleme, Finansal Modelleme, Yapay Zeka Tabanlı Simülasyon ve Oyun Motorları, Veri Madenciliği, Büyük Veri Analitiği, Kripto Madenciliği ve Blockchain gibi birçok alanda kullanılmaktadır.

2. Multi GPU programlama:

Bir veri seti üzerinde işlem yapan bir CUDA Kernel'i farklı GPU'larda çalıştırarak o veri setinin farklı kısımları üzerinde aynı işlemi eş zamanlı yapabiliriz. Ayrıca bir CUDA Kernel'in farklı veri setleri için eş zamanlı çalışmasını da her bir veri setini farklı bir GPU'ya göndererek sağlayabiliriz. Farklı CUDA Kernel'lerini de farklı GPU'larda eş zamanlı çalıştırabiliriz. Bu işlemleri yapabilmek için önce CUDA programlama dilindeki GPU ile alakalı bazı komutları bilmemiz gerekir. Aşağıda multi GPU ile alakalı bazı temel komutlar verilmektedir:

- cudaGetDeviceCount(int *sayi)//CPU'nun erişebildiği cihaz sayısını sayi değişkenine kopyalamaktadır.
- cudaSetDevice(int sayi)//Programın o anki CUDA ile ilgili işlemlerin hangi cihaz üzerinde yapılacağını belirler. Cihazların değeri 0'dan başlayıp ardışık şekilde artarak devam etmektedir ve cihaz sayısı sayi değişkeni ile metoda iletilmektedir.
- cudaGetDevice(int *sayi)//CPU o an hangi cihaz üzerinde CUDA işlemleri yapıyorsa o cihazın değerini sayi değişkenine kopyalamaktadır.
- cudaGetDeviceProperties(cudaDeviceProp *prop, int sayi)//sayi değişkeni ile girilen ilgili cihazın hem yazılımsal hem de donanımsal özellikleri ile ilgili bilgilere erişmek için kullanılmaktadır. prop yapı değişkeni ile bu özelliklerin tutulduğu değişkenlere erişilmektedir.
- cudaDeviceSynchronize()//Cihazdaki kendisinden önceki işlemler tamamlanana kadar CPU'yu bekletir.

Aşağıda satır sayıları GPU sayısına eşit olan CPU'da tutulan iki boyutlu dizilerin her bir satırını farklı bir cihaza gönderen ve üzerinde aynı CUDA Kernel işlemini uygulayıp sonucu CPU'daki üçüncü bir iki boyutlu dizinin ilgili satırına kopyalayan kod parçası verilmektedir:

```
int *sayi;
cudaGetDeviceCount(sayi);
for(int i=0; i<sayi; i++)
{
        cudaSetDevice(i);
        cudaMemcpy(A_GPU[i], A_CPU[i], buyukluk, H2D);
        cudaMemcpy(B_GPU[i], B_CPU[i], buyukluk, H2D);
        kernel<<<n,128>>>(A_GPU[i], B_GPU[i], C_GPU[i]);
        cudaMemcpy(C_CPU[i],C_GPU[i], buyukluk, D2H);
```

```
}
```

Bu örnekte her bir bellek transfer işlemlerinin ve CUDA kernel işleminin tüm GPU'larda eş zamanlı olarak çalışmasını beklemekteyiz. Fakat bellek transfer işlemleri senkron komutlar olduğu için yani ilgili bellek işlemi tamamlanana kadar CPU (host) beklemede kaldığı için işlemler serileşmektedir. Bu yüzden GPU'lar eş zamanlı çalışamamaktadır. cudaMemcpy komutunun asenkron versiyonu olan cudaMemcpyAsync komutunu aşağıdaki gibi kullanarak GPU'ların eş zamanlı çalışmasını sağlayabiliriz:

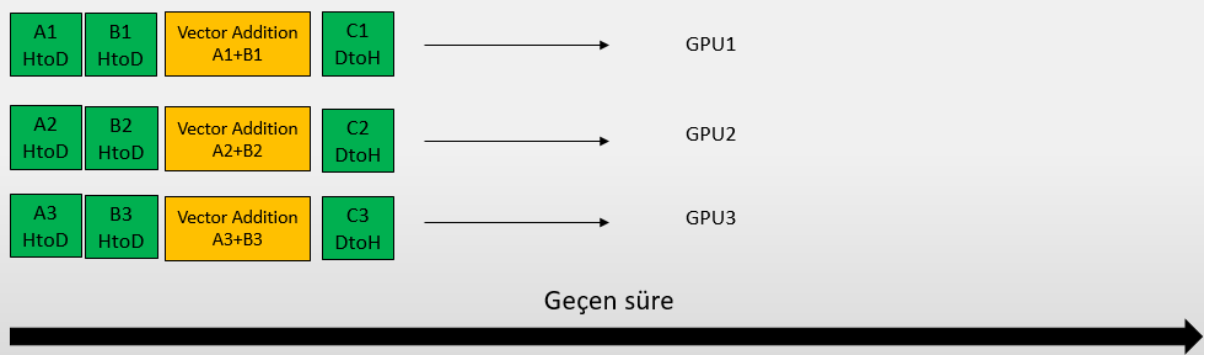
```
int *sayi;
cudaGetDeviceCount(sayi);
for(int i=0; i<sayi; i++)
{
	cudaSetDevice(i);
	cudaMemcpyAsync(A_GPU[i], A_CPU[i], buyukluk, H2D);
	cudaMemcpyAsync(B_GPU[i], B_CPU[i], buyukluk, H2D);
	kernel<<<n,128>>>(A_GPU[i], B_GPU[i], C_GPU[i]);
	cudaMemcpyAsync(C_CPU[i],C_GPU[i], buyukluk, D2H);
}
```

Böylelikle işlemleri erken biten GPU diğerlerini beklemeden programdaki bir sonraki işlemlerini çalıştırmaya başlayabilir. Eğer tüm GPU'ların programdaki sonraki işlemlerine aynı zamanda geçmeleri gerekiyorsa bu GPU'ları senkronize etmemiz gerekir. Onun için de cudaDeviceSynchronize komutunu aşağıdaki gibi kullanabiliriz:

```
int *sayi;
cudaGetDeviceCount(sayi);
for(int i=0; i<sayi; i++)
{
	cudaSetDevice(i);
	cudaMemcpyAsync(A_GPU[i], A_CPU[i], buyukluk, H2D);
	cudaMemcpyAsync(B_GPU[i], B_CPU[i], buyukluk, H2D);
	kernel<<<n,128>>>(A_GPU[i], B_GPU[i], C_GPU[i]);
	cudaMemcpyAsync(C_CPU[i], C_GPU[i], buyukluk, D2H);
}

for(int i=0; i<sayi; i++)
{
	cudaSetDevice(i);
	cudaDeviceSynchronize();
}
```

3. Multi GPU ve CUDA Streams:

CUDA Streams eğitim dokümanında bahsettiğimiz vektör toplama örneğini veri setini üçe bölüp üç ayrı stream oluşturarak tek bir GPU'da eş zamanlı çalıştırmıştık. Ama tek bir GPU olduğu için kaynakları pipelining tekniği ile eş zamanlı kullanmaya çalışmaktaydık. Multi GPU yöntemi ile bu örneğin veri setini üçe bölüp üç ayrı GPU'da eş zamanlı olarak çalıştırabiliriz. Bu durumu şu şekilde gösterebiliriz:



Görüldüğü üzere veri setinin bölünmüş üç farklı parçası üç farklı GPU'da paralel olarak çalışmakta ve süre açısından önemli ölçüde kazanç sağlanmaktadır. Aşağıda bu işlemleri yapan CUDA kodunun bir kısmı verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Multi-GPU Vektör Toplama (Basit) isimli kodu indirebilirsiniz.

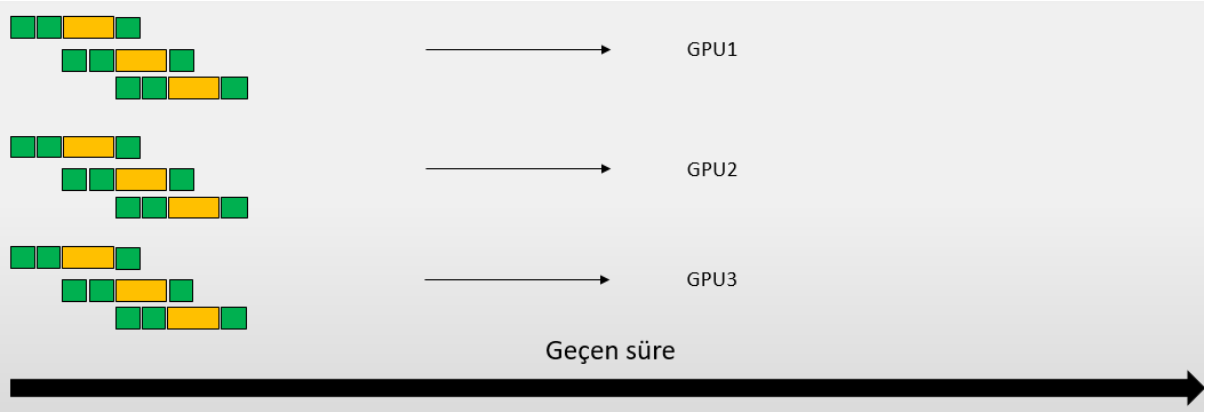
……..

```
cudaSetDevice(0);//İlk GPU aktif oluyor
cudaMemcpyAsync(A1_GPU,A_Host+0*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
cudaMemcpyAsync(B1_GPU,B_Host+0*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
vector_addition<<<DimGrid,DimBlock>>>(A1_GPU,B1_GPU,C1_GPU,size/3);
cudaMemcpyAsync(C_Host+0*(size/3),C1_GPU,sizeof(int)*size/3,cudaMemcpyDeviceToHost);

cudaSetDevice(1);//İkinci GPU aktif oluyor
cudaMemcpyAsync(A2_GPU,A_Host+1*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
cudaMemcpyAsync(B2_GPU,B_Host+1*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
vector_addition<<<DimGrid,DimBlock>>>(A2_GPU,B2_GPU,C2_GPU,size/3);
cudaMemcpyAsync(C_Host+1*(size/3),C2_GPU,sizeof(int)*size/3,cudaMemcpyDeviceToHost);

cudaSetDevice(2);//Üçüncü GPU aktif oluyor
cudaMemcpyAsync(A3_GPU,A_Host+2*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
cudaMemcpyAsync(B3_GPU,B_Host+2*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice);
vector_addition<<<DimGrid,DimBlock>>>(A3_GPU,B3_GPU,C3_GPU,size/3);
cudaMemcpyAsync(C_Host+2*(size/3),C3_GPU,sizeof(int)*size/3,cudaMemcpyDeviceToHost);

cudaSetDevice(0);//İlk GPU aktif oluyor
cudaDeviceSynchronize();//GPU1'deki işlemler bitene kadar host bekliyor
cudaSetDevice(1);//İkinci GPU aktif oluyor
cudaDeviceSynchronize();//GPU2'deki işlemler bitene kadar host bekliyor
cudaSetDevice(2);//Üçüncü GPU aktif oluyor
cudaDeviceSynchronize();//GPU3'teki işlemler bitene kadar host bekliyor
```

……..

cudaSetDevice komutu ile ilgili GPU aktif olmakta ve CPU sonraki komutları aktif olan bu GPU'da çalıştırmaktadır. CUDA Kernel'ler asenkron çalışmaktaydı. cudaMemcpyAsync komutu da asenkron çalıştığı için tüm işlemler üç ayrı GPU'da eş zamanlı çalışmaktadır. cudaDeviceSynchronize komutları ile CPU (host) sonraki komutları çalıştırmak için tüm GPU'lardaki işlemlerin bitmesini beklemektedir. Bu örnekte toplam geçen süreyi hesaplamak istediğimiz için tüm GPU'lardaki işlemlerin bitmesini beklemekteyiz. cudaDeviceSynchronize komutu ile bu beklemeyi sağlamaktayız.

Her bir GPU'da CUDA Stream'leri kullanarak daha da hızlanma elde edebiliriz. Her bir GPU'da yer alan veri setinin bölünmüş parçalarını da üçe bölelim ve üç ayrı stream oluşturalım:



Bu yaklaşım ile ekstra hızlanma elde etmiş olmaktayız. Aşağıda bu işlemleri yapan CUDA kodunun bir kısmı verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Multi-GPU Vektör Toplama (Streams) isimli kodu indirebilirsiniz.

……..

```
for(int i=0;i<3;i++)//Her bir GPU için işlem yapacak
{
	cudaSetDevice(i);//İlgili GPU aktif oluyor

	//Parçaların işlemleri CUDA Stream'ler kullanılarak overlap ediliyor

	cudaMemcpyAsync(A1_GPU[i],A_Host+((i*3)+0)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][0]);
	cudaMemcpyAsync(B1_GPU[i],B_Host+((i*3)+0)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][0]);
	vector_addition<<<DimGrid,DimBlock,0,stream[i][0]>>>(A1_GPU[i],B1_GPU[i],C1_GPU[i],size/9);
	cudaMemcpyAsync(C_Host+((i*3)+0)*(size/9),C1_GPU[i],sizeof(int)*size/9,cudaMemcpyDeviceToHost,stream[i][0]);

	cudaMemcpyAsync(A2_GPU[i],A_Host+((i*3)+1)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][1]);
	cudaMemcpyAsync(B2_GPU[i],B_Host+((i*3)+1)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][1]);
	vector_addition<<<DimGrid,DimBlock,0,stream[i][1]>>>(A2_GPU[i],B2_GPU[i],C2_GPU[i],size/9);
	cudaMemcpyAsync(C_Host+((i*3)+1)*(size/9),C2_GPU[i],sizeof(int)*size/9,cudaMemcpyDeviceToHost,stream[i][1]);

	cudaMemcpyAsync(A3_GPU[i],A_Host+((i*3)+2)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][2]);
	cudaMemcpyAsync(B3_GPU[i],B_Host+((i*3)+2)*(size/9),sizeof(int)*size/9,cudaMemcpyHostToDevice,stream[i][2]);
	vector_addition<<<DimGrid,DimBlock,0,stream[i][2]>>>(A3_GPU[i],B3_GPU[i],C3_GPU[i],size/9);
```

```
	cudaMemcpyAsync(C_Host+((i*3)+2)*(size/9),C3_GPU[i],sizeof(int)*size/9,cudaMemcpyDevi
ceToHost,stream[i][2]);

}

for(int i=0;i<3;i++)//Her bir GPU için işlem yapacak
{
	cudaSetDevice(i);//İlgili GPU aktif oluyor
	cudaStreamSynchronize(stream[i][0]);//1. stream'deki işlemler bitene kadar host bekliyor
	cudaStreamSynchronize(stream[i][1]);//2. stream'deki işlemler bitene kadar host bekliyor
	cudaStreamSynchronize(stream[i][2]);//3. stream'deki işlemler bitene kadar host bekliyor
}
……..
```

Her bir GPU için ayrı ayrı kod yazmak yerine döngü içinde ilgili komutları tek sefer yazmamız yeterlidir. Tüm GPU'ların birbirinden bağımsız şekilde çalışması gerektiği için bellek işlemlerinde kullanılan işaretleyicilerin (pointer) ve stream değişkenlerinin her bir GPU'ya özgü tanımlanması gerekir. Bu yüzden bu işaretleyiciler ve stream'ler iki boyutlu dizi olarak tanımlandı. Tek GPU'lu örneklerde olduğu gibi her bir GPU'daki stream'ler için cudaStreamSynchronize komutları kullanılarak host'un tüm GPU'lardaki işlemlerin tamamlanmasını beklemesi sağlandı.

4. OpenMP kullanarak Multi GPU yaklaşımı:

Her ne kadar asenkron çalışan komutlar kullansak bile host'un her GPU için ilgili işlemleri başlatması seri şekilde olmaktadır. Tüm GPU'ların tam olarak aynı anda işlemlerine başlayabilmesi için OpenMP uygulama geliştirme ara yüzünü kullanabiliriz. OpenMP birden çok işlemciye sahip bilgisayarların her bir işlemcisi için iş parçacığı (thread) oluşturarak onların eş zamanlı kullanılmasını sağlamaktadır. Son örneğimizde her bir GPU için bir OpenMP iş parçacığı (thread) tanımlayarak tüm GPU'lardaki işlemlerin tam olarak eş zamanlı gerçekleştirilmesini sağlayabiliriz. Aşağıda bu yaklaşım için yazılmış CUDA kodunun bir kısmı verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Multi-GPU Vektör Toplama (OpenMP) isimli kodu indirebilirsiniz.

```
……..

double start = omp_get_wtime();//Başlangıç zamanı alınıyor

#pragma omp parallel private(tid)//OpenMP paralel blok başlıyor
{
	tid = omp_get_thread_num();//Thread id'si alınıyor
	cudaSetDevice(tid);//İlgili GPU aktif oluyor

	//Parçaların işlemleri CUDA Stream'ler kullanılarak overlap ediliyor

	cudaMemcpyAsync(A1_GPU[tid],A_Host+((tid*3)+0)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][0]);
	cudaMemcpyAsync(B1_GPU[tid],B_Host+((tid*3)+0)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][0]);
	vector_addition<<<DimGrid,DimBlock,0,stream[tid][0]>>>(A1_GPU[tid],B1_GPU[tid],C1_GP
U[tid],size/9);
```

```
        cudaMemcpyAsync(C_Host+((tid*3)+0)*(size/9),C1_GPU[tid],sizeof(int)*size/9,cudaMemcpy
DeviceToHost,stream[tid][0]);

        cudaMemcpyAsync(A2_GPU[tid],A_Host+((tid*3)+1)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][1]);
        cudaMemcpyAsync(B2_GPU[tid],B_Host+((tid*3)+1)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][1]);
        vector_addition<<<DimGrid,DimBlock,0,stream[tid][1]>>>(A2_GPU[tid],B2_GPU[tid],C2_GP
U[tid],size/9);
        cudaMemcpyAsync(C_Host+((tid*3)+1)*(size/9),C2_GPU[tid],sizeof(int)*size/9,cudaMemcpy
DeviceToHost,stream[tid][1]);

        cudaMemcpyAsync(A3_GPU[tid],A_Host+((tid*3)+2)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][2]);
        cudaMemcpyAsync(B3_GPU[tid],B_Host+((tid*3)+2)*(size/9),sizeof(int)*size/9,cudaMemcpy
HostToDevice,stream[tid][2]);
        vector_addition<<<DimGrid,DimBlock,0,stream[tid][2]>>>(A3_GPU[tid],B3_GPU[tid],C3_GP
U[tid],size/9);
        cudaMemcpyAsync(C_Host+((tid*3)+2)*(size/9),C3_GPU[tid],sizeof(int)*size/9,cudaMemcpy
DeviceToHost,stream[tid][2]);

        cudaStreamSynchronize(stream[tid][0]);
        cudaStreamSynchronize(stream[tid][1]);
        cudaStreamSynchronize(stream[tid][2]);
}

double end = omp_get_wtime();//Bitiş zamanı alınıyor
printf("Toplam Süre = %f saniye\n",end - start);//Geçen süre hesaplanıyor
printf("C[size-1] = %d\n",C_Host[size-1]);

……..
```

Bu örnekte OpenMP kullanılarak her bir GPU için bir iş parçacığı (thread) oluşturulup ölçümlenen işlemlerin tamamen eş zamanlı çalışması sağlanmaktadır. Böylelikle önceki GPU'lardaki komutların çalıştırılmaya başlanması için geçen süreler sonraki GPU'lardaki komutların çalıştırılmaya başlanmasını geciktirmemektedir. Bu yaklaşımla program çalışma süresi açısından önemli düzeyde kazanç sağlanmaktadır. #pragma omp parallel private(tid) direktifi ile her bir iş parçacığının (thread) eş zamanlı çalışmasını sağlayan paralel bir blok oluşmaktadır. Bu paralel bloktaki komutları her bir iş parçacığı (thread) birbirinden bağımsız şekilde çalıştırır. Thread id bilgisi kullanılarak her bir iş parçacığının farklı bir GPU'yu aktif etmesi (cudaSetDevice(tid) komutu ile) ve o GPU ile ilgili hesaplamaların yapılması sağlanmıştır. Bu yaklaşımda süreleri ölçmek için kullanılan cudaEventRecord metodu thread-safe bir metot olmadığı için doğru sonuç vermemektedir. Onu yerine OpenMP'de yer alan ve thread-safe olan omp_get_wtime metodu kullanılmıştır.

5. Multi-GPU ile ilgili bazı önemli bilgiler:

GPU’lar aralarında direk (P2P – Peer to Peer) bir veri aktarımı yapabilir. Bu sayede daha verimli bir iletişim oluşmakta ve CPU’nun belleğine gitmeden hızlı veri transferi sağlanmaktadır. Bunun için İlk

olarak iki GPU arasında P2P erişim aktif hale getirilmelidir. Aşağıda bu konuyla alakalı bazı metotlar verilmektedir:

- cudaDeviceEnablePeerAccess (int peerDevice, 0) metodu ile P2P erişim aktif hale getirilmektedir.
- cudaDeviceDisablePeerAccess (int peerDevice) metodu ile P2P erişim pasif hale getirilmektedir.
- cudaDeviceCanAccessPeer (int* canAccessPeer, int device, int peerDevice) metodu P2P erişim var mı kontrolü yapmaktadır (0-yok, 1-var).
- cudaMemcpyPeer (void* dst, int dstDevice, const void* src, int srcDevice, size_t count) metodu iki GPU arasında P2P veri transferi için kullanılmaktadır.

Bazı ek kavramlara da aşağıda değinilmektedir:

- Unified Memory kavramı tüm GPU'ların ve CPU'nun aynı sanal belleği paylaşması ile ilgilidir.
- NVLink kavramı GPU'lar arası yüksek hızlı bağlantı sağlamaktadır.
- GPUDirect RDMA kavramı GPU belleğine doğrudan ağ kartı üzerinden erişim ile ilgilidir.
- Multi-Process Service (MPS) kavramı birden fazla uygulamanın aynı anda tek bir GPU'yu paylaşmasını sağlamaktadır.
- NCCL (NVIDIA Collective Comm. Library) kavramı GPU'lar arası yüksek performansta veri paylaşımı ile ilgilidir.