

# CUDA STREAMS

## 1. Single instruction multiple threads (SIMT) modeli:

Tek komut çoklu veri (Single instruction multiple data (SIMD)) paralel işleme modelinde bir bilgisayardaki birden çok işlemci aynı işlemi birden fazla veri noktası üzerinde eş zamanlı olarak işlemektedir. Bu tür bilgisayarlar veri düzeyinde paralellikten (Data level parallelism) yararlanırlar. Veri düzeyinde paralellikte veriler farklı işlemcilere dağıtılarak verinin her bir parçasında aynı işlem eş zamanlı olarak gerçekleştirilmektedir. Diziler ve matrislerin her bir elemanında aynı anda aynı işlemi yapmak için bu model oldukça uygundur. Fakat bu modelin eş zamanlılıktan tam olarak yararlandığından bahsedemeyiz. Çünkü aynı anda farklı işlemciler tarafından farklı veri noktaları üzerinde işlemler yapılsa da tüm işlemciler aynı anda tamamen aynı olan bir komutu çalıştırmaktadırlar. SIMD modelinin çoklu iş parçacığı (multithreading) ile birleşiminden tek komut çoklu iş parçacıkları (Single instruction multiple threads (SIMT)) paralel işleme modeli ortaya çıkmıştır. Bu model birden çok yazılımsal yada donanımsal iş parçacıklarının (thread) görevlerini yerine getirmek için aynı anda aynı işlemciyi zaman dilimleme (time-slicing) şeklinde paylaşılarak kullanmaları ile karıştırılmamalıdır. SIMT modeli donanım düzeyinde paralellik sağlayan gerçek anlamda bir eş zamanlılık sunmaktadır. Komutların tüm iş parçacıkları tarafından kilit adım (lock-step) şekliyle yürütülmesiyle SIMT modeli tek program çoklu veri (Single program multiple data (SPMD)) modelinden farklılaşmaktadır. Yani SIMT modelinde iş parçacıkları tarafından aynı anda farklı veri noktaları için çalıştırılan bir komut tüm iş parçacıkları tarafından tamamlanmadan program bir sonraki komutun çalıştırılmasına geçmemektedir. Bu model bazı grafik işleme birimlerinde (GPU) uygulanmıştır ve grafik işleme birimlerinde genel amaçlı hesaplamalar (GPGPU) yapmak için çok uygundur. CUDA GPU programlama dili SIMT paralel işleme modelini uygulamıştır. GPU üzerinde fiziksel olarak yer alan warp'lar GPU çekirdeklerini gerçek anlamda eş zamanlı olarak kullanmaktadırlar. Her warp'ta yer alan 32 tane iş parçacığı birbirleriyle fiziksel olarak bağlıdırlar. Bu iş parçacıklarının hepsi farklı veri noktaları için çalıştırdıkları komutu çalıştırmayı bitirmeden bir sonraki komuta geçememektedirler. Böylelikle iş parçacıkları arasında kilit adım (lock-step) uygulanmış olmaktadır.

## 2. CUDA Stream Giriş:

Birbirinden bağımsız görevlerin (task) eş zamanlı çalışmasına görev düzeyinde paralellik (Task level parallelism) denilmektedir. Görev paralelliği süreçler (process) yada iş parçacıkları tarafından eş zamanlı çalıştırılan farklı görevlerin farklı işlemcilere dağıtılmasına odaklanmaktadır. Veri paralelliği bir verinin değişik noktaları için aynı görevi çalıştırmakla ilgilenirken görev paralelliği aynı veri üzerinde birçok farklı görevin eş zamanlı çalışmasıyla ilgilenir. CUDA programlama dilinde görev paralelliği CUDA Stream'ler kullanılarak sağlanmaktadır. CUDA stream'ler ile ilgili bazı bilgiler şu şekildedir:

- Aynı stream'lerdeki komutlar seri olarak çalışmaktadır. İlk giren ilk çıkar (First in first out (FIFO)) ilkesi geçerlidir.
- Bir stream'deki komutlar diğer stream'deki komutlarla birlikte eş zamanlı çalışabilir.
  - Fakat merkezi işlem birimi (CPU) ile GPU arasındaki veri transferi işlemlerinde herhangi bir yönde aynı anda sadece bir stream transfer işlemi gerçekleştirilebilir.
  - Biri CPU'dan GPU'ya diğeri GPU'dan CPU'ya olmak üzere aynı anda en fazla iki veri transfer işlemi gerçekleştirilebilir.
- Her bir CUDA programında bir varsayılan (default) stream (stream 0) vardır. Sonradan oluşturulan stream'lerin komutları bu varsayılan stream'in komutları ile eş zamanlı

çalıştırılmaz. Dolayısıyla eş zamanlı çalıştırmak istediğimiz tüm işlemler için ayrı bir stream oluşturmamız gerekir.

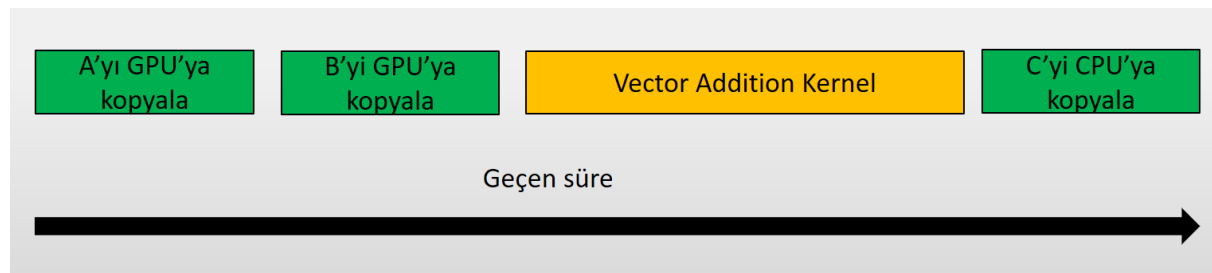
- Aşağıda verilen bazı CUDA komutları senkronize çalışan komutlardır. Yani program akışı GPU'daki (device) işlemler bittikten sonra tekrar CPU'ya (host) dönmektedir.
  - `cudaMalloc`
  - `cudaMemcpy`
  - `cudaMallocHost`
  - `cudaFree`
- Her CUDA işleminden önce ve sonra `cudaDeviceSynchronize` komutunu kullanırsak host (CPU) ile device (GPU) arasındaki tüm işlemler tamamen senkron olmaktadır. Yani asenkron çalışan CUDA komutlarını kullansak bile program akışı host'a geçmeden önce device'daki tüm işlemlerin tamamlanması beklenmektedir.
- Birden çok stream'deki işlemlerin eş zamanlı çalışmasını istiyorsak host'a karşı asenkron çalışan CUDA komutlarını kullanmamız gerekir. Bu asenkron çalışan CUDA komutlarından bazıları şunlardır:
  - CUDA Kernel
  - `cudaMallocAsync`
  - `cudaMemcpyAsync`
  - `cudaMemsetAsync`
  - `cudaEventRecord`
- Aşağıdaki bazı asenkron CUDA komutları farklı stream'ler kullanılarak eş zamanlı çalıştırılabilir:
  - CUDA Kernel <<<>>>
  - `cudaMemcpyAsync (HostToDevice)`
  - `cudaMemcpyAsync (DeviceToHost)`
  - CPU'daki işlemler
- Hesaplama yeteneği (Compute capability) 2.0 ve üzeri olan mimarilerde aşağıdaki işlemler eş zamanlı olarak çalıştırılabilir:
  - Aynı anda 16 CUDA kernel
  - 2 `cudaMemcpyAsync` (bir tanesi `HostToDevice`, diğeri `DeviceToHost`) komutu
  - CPU'daki işlemler
- CUDA ile görevleri eş zamanlı çalıştırmak için aşağıdaki gereksinimlerin sağlanması gerekir:
  - CUDA işlemleri stream 0 dışındaki stream'lerde çalıştırılması gerekir.
  - `cudaMemcpyAsync` komutu kullanarak host ile veri alış verişi yaparken sabitlenmiş bellek (Pinned memory) kullanılmalıdır. Bu belleğe ayrıca sayfa kilitli bellekte (Page-locked memory) denir.
    - `cudaMallocHost()` yada `cudaHostAlloc()` komutları kullanılarak oluşturulmaktadır.
    - Ayrıca bu bellek bölgesi host'a ait diske hiçbir zaman kaydedilmez. Böylelikle bu bellek bölgesi her zaman erişilebilir olur ve host ile device arasında hızlı bir veri alış verişi sağlar.
  - Yeterince kaynak kullanılabilir olmalıdır:
    - `cudaMemcpyAsync` komutu her iki yön için kullanılabilir olmalıdır.
    - Ortak bellek (shared memory), yazmaçlar (register), bloklar (block) gibi GPU kaynaklarının yeterli bir miktarda kullanılabilir durumda olması gerekir.

### 3. Vektör Toplama Örneği:

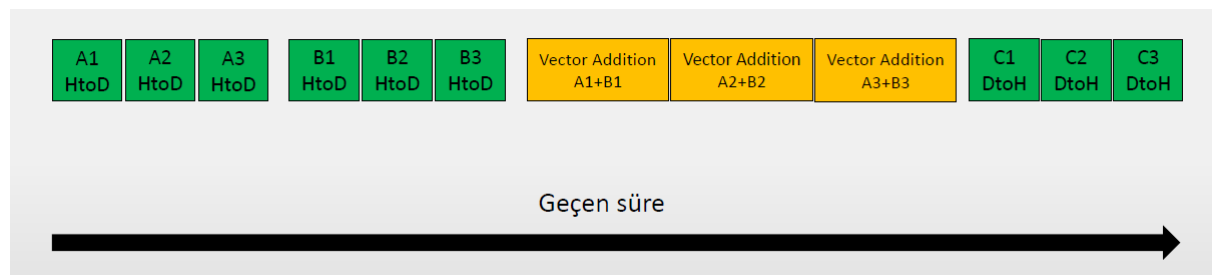
Daha önce gördüğümüz vektörleri toplama problemini hatırlayalım. Tek boyutlu iki vektörün aynı indeksteki elemanlarını toplayıp üçüncü bir vektördeki aynı indekse kaydetmekteydik. Bu problemin CUDA kodunun bir kısmı aşağıda verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Vektör Toplama (Basit) isimli kodu indirebilirsiniz.

```
.....  
cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);  
cudaMemcpy(B_GPU,B_Host,sizeof(int)*size,cudaMemcpyHostToDevice);  
vector_addition<<<DimGrid,DimBlock>>>(A_GPU,B_GPU,C_GPU,size);  
cudaMemcpy(C_Host,C_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);  
.....
```

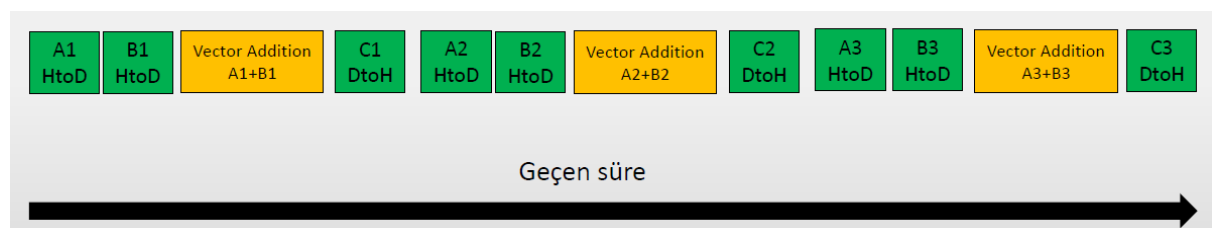
Varsayılan stream'i (stream 0) kullandığımız bu örnekte cudaMemcpy ve CUDA kernel çalıştırma işlemleri seri olarak çalışmaktadır. Yani veri transferi yapılırken GPU çekirdekleri boşa kalmakta, CUDA kernel çalışırken de veri transfer yolu boşa kalmaktadır. Aşağıda bu durum görsel olarak verilmektedir.



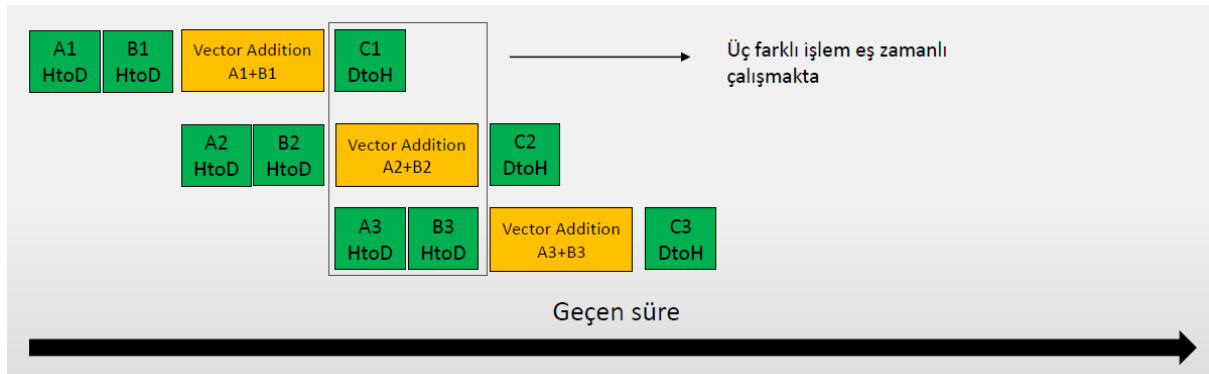
Şimdi borulama (pipelining) tekniği ve CUDA stream'ler kullanarak görev paralelliği elde edip aynı işlemi daha hızlı bir şekilde çözmeyi hedefliyoruz. Öncelikle veri setini küçük parçalara ayıralım. Yukarıdaki örnekteki **A**, **B** ve **C** vektörlerindeki verileri üçe bölelim (**A1**, **A2**, **A3** gibi):



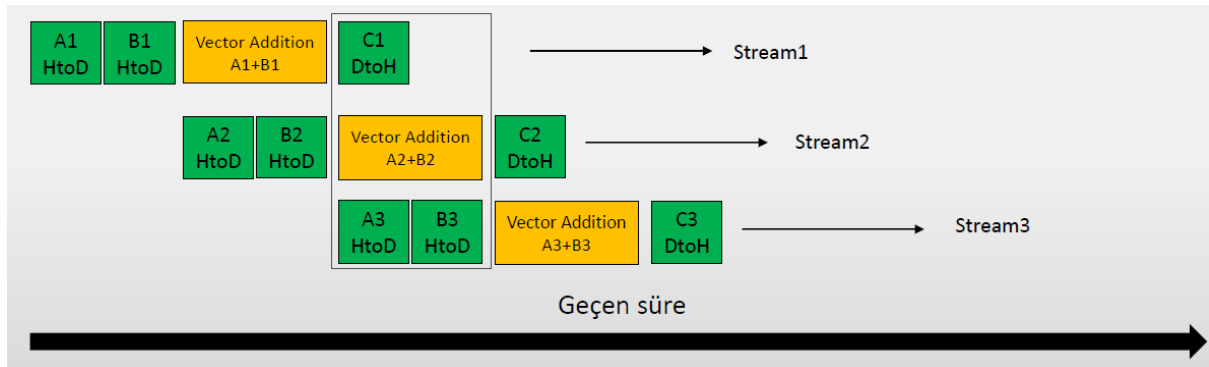
İşlemlerin yerlerini değiştirerek aşağıdaki gibi üç tane bağımsız vektör toplama işlemi elde ediyoruz. Böylelikle veri transferleri ile CUDA kernel işlemlerini örtüştürebiliriz.



Bağımsız işlemleri örtüştürmek için pipelining tekniğini kullanacağız. Sonrasında CUDA stream'leri kullanarak eş zamanlı çalışmalarını sağlayacağız. Pipelining kullanarak elde ettiğimiz yapı şu şekilde olmaktadır:



Aynı yöndeki bellek transfer işlemleri (**A1** ve **B1**'in transferi gibi) seri olarak çalışmaya devam etmektedir. Bellek transferleri ile CUDA kernel'lerinin çalışmaları da seri olacaktır. Fakat farklı görevlerin bellek transfer işlemleri ile CUDA kernel'lerini örtüştürebiliriz. Bu örnekte görüldüğü üzere **A1** ve **B1**'in vektör toplama işlemi ile **A2** ve **B2**'in bellek transfer işlemleri aynı anda gerçekleştirilmektedir. Aynı durum **A2**, **B2** ile **A3**, **B3** arasında da geçerlidir. Ayrıca iki farklı yöndeki bellek transferleri de örtüşebilir. Bu örnekte görüldüğü üzere **C1**'in değerleri GPU'dan CPU'ya transfer edilirken aynı anda **A3** ve **B3**'ün değerleri CPU'dan GPU'ya transfer edilmektedir. Ayrıca **A2** ve **B2**'nin vektör toplama işlemi de bu transfer işlemleri ile aynı anda çalışmaktadır. Böylelikle üç farklı türde bağımsız işlemi örtüştürerek aynı anda çalışmalarını sağlayıp programın çalışma süresinde önemli kazançlar elde etmiş olacağız. Her bir küçük bağımsız vektör toplama işlemi için ayrı bir stream oluşturarak yukarıda bahsettiğimiz işlemlerin eş zamanlı olarak çalışmasını sağlayacağız.



İşlemlerin eş zamanlı çalışmasını sağlayabilmek için senkron komutların (cudaMemcpy gibi) asenkron versiyonlarını kullanmamız gerekir. CUDA kernel'ler host'a göre zaten asenkron olduğu için aynen alıp kullanabiliriz. CPU GPU arasındaki transferlerin hızlı yapılabilmesi için CPU'daki veriler için "pinned memory" şeklinde bellek ayırma işlemi yapmamız gerekir. Aşağıda bu işlemleri yapan CUDA kodunun bir kısmı verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Vektör Toplama (Streams) isimli kodu indirebilirsiniz.

.....

```
cudaMallocHost((void**)&A_Host, sizeof(int)*size);//CPU belleğinde (Pinned) yer açılıyor
cudaMallocHost((void**)&B_Host, sizeof(int)*size);//CPU belleğinde (Pinned) yer açılıyor
cudaMallocHost((void**)&C_Host, sizeof(int)*size);//CPU belleğinde (Pinned) yer açılıyor
```

.....

```
cudaStream_t stream[3];
cudaStreamCreate(&stream[0]); //1.stream oluşturuluyor
cudaStreamCreate(&stream[1]); //2.stream oluşturuluyor
cudaStreamCreate(&stream[2]); //3.stream oluşturuluyor
.....
```

//Parçaların işlemleri CUDA Stream'ler kullanılarak overlap ediliyor

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

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

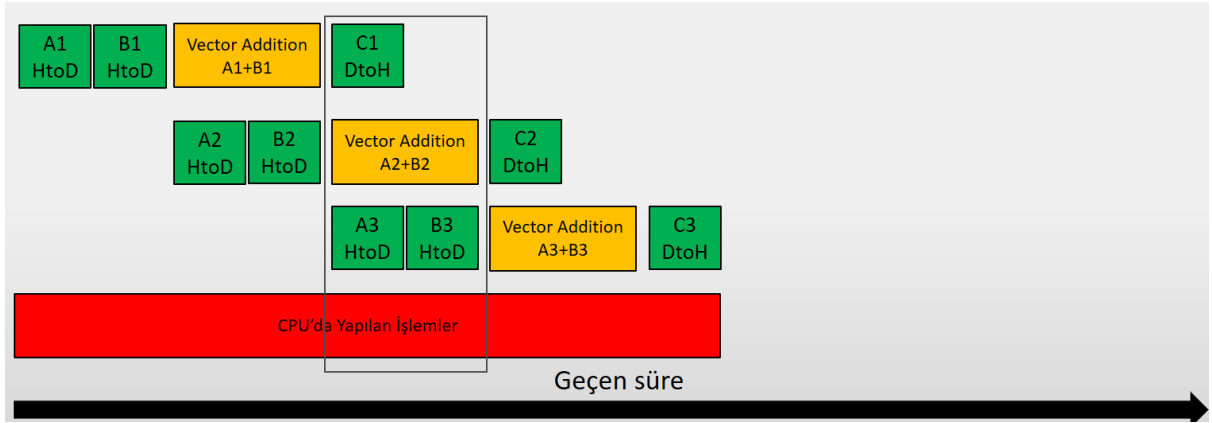
```
cudaMemcpyAsync(A3_GPU,A_Host+2*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice,stream[2]);
cudaMemcpyAsync(B3_GPU,B_Host+2*(size/3),sizeof(int)*size/3,cudaMemcpyHostToDevice,stream[2]);
vector_addition<<<DimGrid,DimBlock,0,stream[2]>>>(A3_GPU,B3_GPU,C3_GPU,size/3);
cudaMemcpyAsync(C_Host+2*(size/3),C3_GPU,sizeof(int)*size/3,cudaMemcpyDeviceToHost,stream[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

```
cudaStreamSynchronize(stream[0]); //Stream 1'deki işlemler bitene kadar host bekliyor
cudaStreamSynchronize(stream[1]); //Stream 2'deki işlemler bitene kadar host bekliyor
cudaStreamSynchronize(stream[2]); //Stream 3'deki işlemler bitene kadar host bekliyor
```

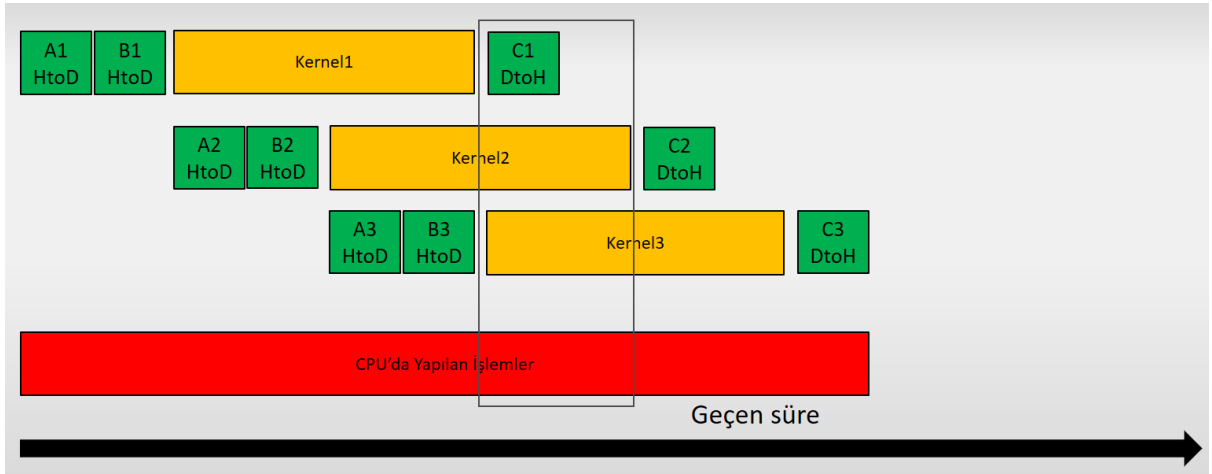
```
.....
cudaStreamDestroy(stream[0]); //Stream 1 yok ediliyor
cudaStreamDestroy(stream[1]); //Stream 2 yok ediliyor
cudaStreamDestroy(stream[2]); //Stream 3 yok ediliyor
.....
```

cudaMallocHost komutu ile ilgili vektörler için CPU'da pinned bellek bölgesi oluşturuyoruz. cudaStreamCreate fonksiyonu ile üç farklı stream oluşturuyoruz. Her bir stream vektörlerin elemanlarının üçte birini alarak aralarında paylaşmaktadır. Bellek transferleri için cudaMemcpy komutunun asenkron versiyonu olan cudaMemcpyAsync komutunu kullanıyoruz. Böylelikle ilgili stream'in işlemleri önceki stream'in işlemleri bitmeden hemen başlayabilmektedir. CUDA kernel çalıştırma işlemleri host'a göre zaten asenkron olduğun için ekstra bir şey yapmamıza gerek olmamaktadır. Hem cudaMemcpyAsync fonksiyonunu çalıştıracığımız zaman hem de CUDA kernel'i çalıştıracığımız zaman hangi stream için çalıştıracığımızı ilgili parametrede doğru bir şekilde belirtmemiz gerekmektedir. cudaStreamSynchronize komutu ile ilgili stream'deki tüm işlemler bitene kadar host'un beklemesini sağlamaktayız. cudaStreamDestroy komutu ile ilk başta yarattığımız stream'leri yok etmekteyiz.

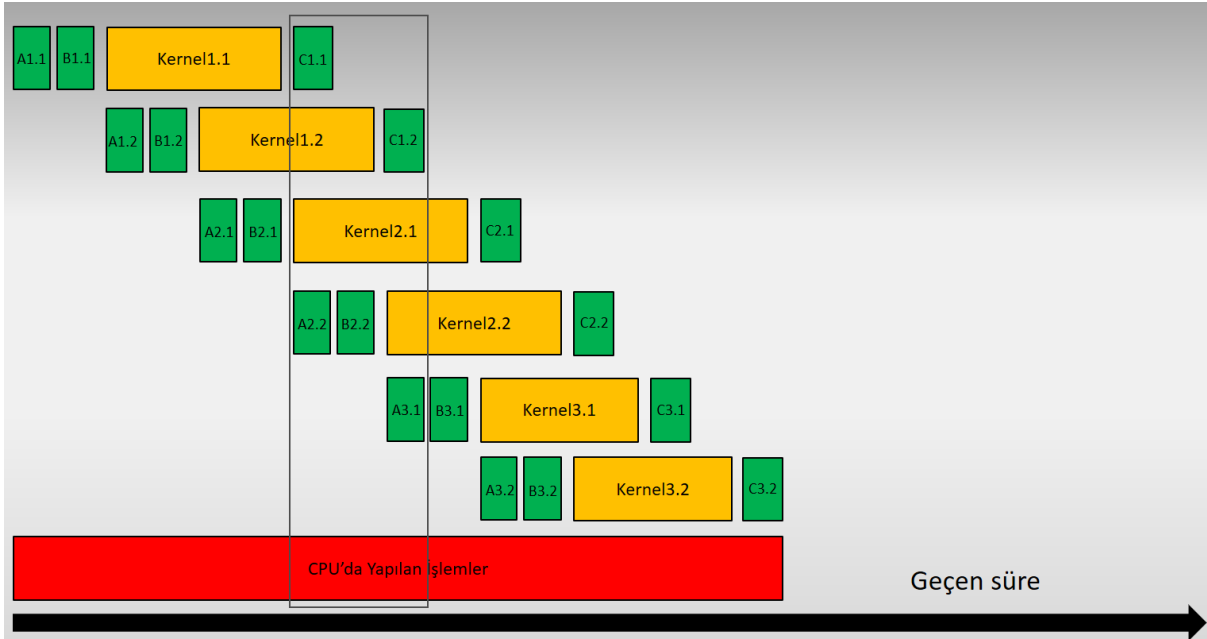
Yukarıdaki örneğimizde 3-yönlü eşzamanlılık (3-way concurrency) elde ettik. GPU'daki işlemleri CPU'daki işlemlerle eş zamanlı çalıştırarak aşağıdaki gibi 4-yönlü eş zamanlılık kazanabiliriz:



Şimdiki örnekte vektör toplama işleminden iki kat daha uzun çalışan bir CUDA kernel düşünelim. A,B ve C dizilerinin veri transfer süreleri ise aynı kalmış olsun. Yine aşağıdaki gibi bir 4-yönlü eş zamanlılık elde edebiliriz:



Bu örnek ile aynı stream'lerdeki CUDA kernel'leri eş zamanlı çalıştırmamız ta farklı stream'lerdeki CUDA kernel'leri eş zamanlı olarak çalıştırabildiğimizi görmekteyiz. Yukarıdaki 4-yönlü eş zamanlılığı CUDA kernel'lerinin işlemlerini daha da küçük parçalara ayırarak daha da arttırabiliriz. Her bir stream'deki işlemleri birbirinden bağımsız iki küçük parçaya ayıralım. Böylelikle aşağıdaki gibi birbirinden bağımsız hesaplama yapacak altı küçük işlem elde etmiş olacağız.



Bu örnekte her bir bağımsız işlem için birer stream oluşturarak 5-yönlü eş zamanlılık elde etmiş olmaktadır. İkiye ayırdığımız CUDA kernel'in her bir parçasını eş zamanlı çalıştırarak çalışma süresini 4-yönlü eş zamanlılık senaryosuna göre daha da iyileştirmiş olduk. GPU'daki işlemler için eş zamanlılığı 4-yönlü eş zamanlılık ve üzerine çıkarmak için CUDA kernel'leri ne kadar çok bağımsız küçük CUDA kernel'lere ayırabilirsek o kadar çok eş zamanlılık elde edebiliriz. Fermi mimarisi 16-yönlü eş zamanlılığa, Kepler mimarisi 32-yönlü eş zamanlılığa kadar izin vermektedir.

#### 4. Stream Scheduling:

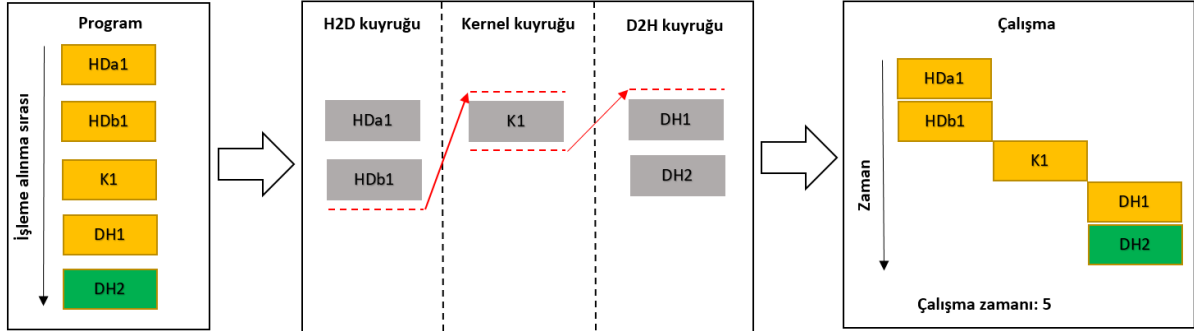
Fermi mimarisinde stream planlama (stream scheduling) yapılırken üç tane kuyruk (queue) işlem yapmaktadır. Bir tanesi CUDA kernel planlamasından sorumlu Hesaplama Motoru kuyruğudur (Compute Engine queue). Diğerleri CPU'dan GPU'ya veri transferi ve GPU'dan CPU'ya veri transferi işlemlerinin planlanmasından sorumlu olan iki tane Kopyalama Motoru kuyruğudur (Copy Engine queue). CUDA işlemleri programdaki işleme alınma sıralarına göre donanımda çalışmak üzere kuyruklara gönderilir. Her işlem kendisi ile alakalı olan kuyruğa yerleştirilir. Bir CUDA işlemi aşağıdaki koşullar sağlandığı zaman çalıştırılmak üzere donanıma gönderilir:

- Aynı stream'deki kendisinden önceki tüm işlemlerin donanımda çalışmasının tamamlanması gerekmektedir.
- Aynı kuyruktaki kendisinden önce yerleştirilen tüm işlemlerin çalıştırılmak üzere donanıma gönderilmesi gerekmektedir.
- Donanımda çalıştırılabilmesi için yeterince kaynağın var olması gerekmektedir.

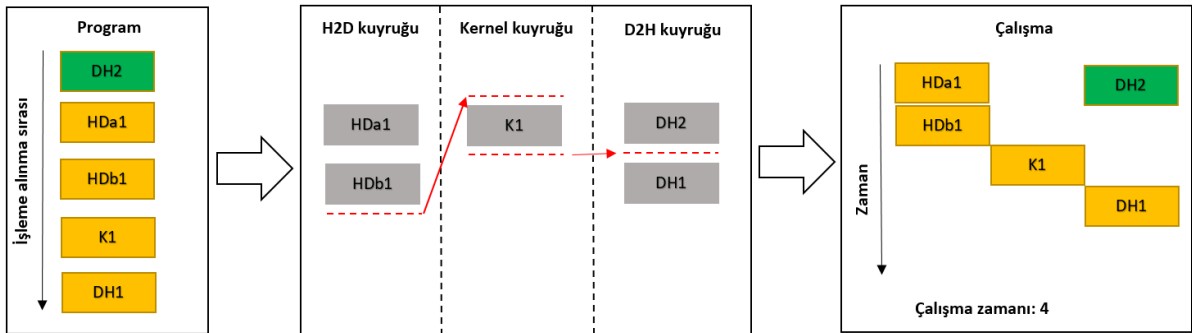
Farklı stream'lerdeki CUDA kernel'lerinin eş zamanlı çalışabileceğini daha önce belirtmiştik. Bu CUDA kernel'lerin eş zamanlı çalışabilmesi için kendisinden önce Hesaplama Motoru kuyruğuna eklenen CUDA kernel'lerinin çalışmak üzere donanıma gönderilmesi gerekmekte ve çoklu işlemci modüllerinde (streaming multiprocessor (SM) ) yeterince kaynağın erişilebilir olması gerekmektedir. Doğru bir stream planlaması yapılabilmesi için CUDA kodumuzu yazarken önemli bir noktaya dikkat etmemiz gerekir. Bir bloklanmış CUDA işlemi kuyruktaki kendisinden sonra gelen tüm işlemlerin –farklı bir stream'de olsa dahi- çalıştırılmak üzere donanıma gönderilmesini engellemektedir. Örnek olarak iki tane stream oluşturalım. Stream'ler aşağıdaki gibi işlemlere sahip olsun:

- Stream 1: HDa1, HDb1, K1, DH1 (Sırayla işleme alınıyor)
- Stream 2: DH2 (Stream 1'deki işlemlerden tamamen bağımsız çalışıyor)

Bu örnekte önce stream 1'deki işlemler programda sırayla işleme alınsın. Sonrasında stream 2'deki işlem programda işleme alınsın. Aşağıdaki şekilde bu işlemlerin programda işleme alınma sıraları, kuyruklara eklenme sıraları ve donanım üzerinde çalışma sıraları gösterilmektedir:



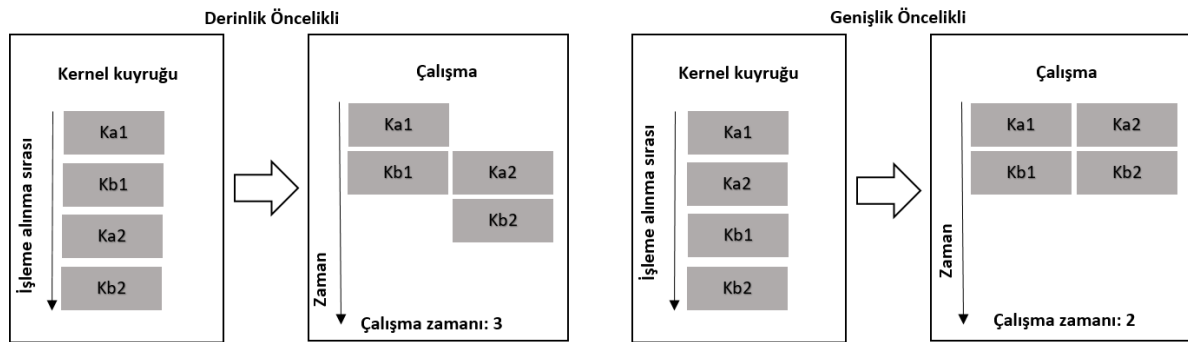
Görüldüğü üzere CUDA işlemleri işleme alınma sıralarına göre ilgili kuyruğa eklenmektedir. Kırmızı oklar ile gösterilen kuyruklar arası sinyaller kuyruklar arasında senkronizasyona sebep olmaktadır. HDb1'in çalıştırılması bitmeden K1 çalışmaya başlayamaz. K1'in çalıştırılması bitmeden DH1 çalışmaya başlayamaz. Aynı kuyruk içindeki farklı stream'lere ait işlemler kendinden önceki işlemlerden bağımsız olsa bile donanımda çalışabilmesi için kuyruktaki bulunduğu sıraya göre kendinden önceki işlemlere bağımlı olmaktadır. DH2 stream 1'deki tüm işlemlerden bağımsız olmasına rağmen kuyruğunda kendisinden önce yer alan DH1'in çalışıp tamamlanmasını beklemek zorunda kalmaktadır. Böylelikle DH1 stream 1'deki tüm işlemlerden bağımsız çalışan DH2'yi bloklayıp onun stream 1'deki işlemlerle eş zamanlı çalışmasını engellemektedir. Bu örnekteki sorunu işlemlerin programda işleme alınma sırasını değiştirerek çözebiliriz. Bu sefer stream 2'deki işlem stream 1'deki işlemlerden önce program tarafından işleme alınsın. Aşağıdaki şekilde bu işlemlerin programda işleme alınma sıraları, kuyruklara eklenme sıraları ve donanım üzerinde çalışma sıraları gösterilmektedir:



Görüldüğü üzere DH1 artık DH2'nin çalışmasını engellemiyor. Böylelikle stream 2'deki DH2 işlemi stream 1'deki işlemlerle eş zamanlı çalışması sağlanmış olmaktadır. Şimdi farklı stream'lerde sadece CUDA kernel işlemlerinin yapıldığı bir örneği inceleyelim. Yine iki tane stream oluşturmuş olalım. Aşağıdaki gibi işlemlere sahip olsunlar:

- Stream 1: Ka1, Kb1
- Stream 2: Ka2, Kb2
- Kernel büyüklükleri benzer olsun ve her bir kernel SM kaynaklarının yarısını kullansın.

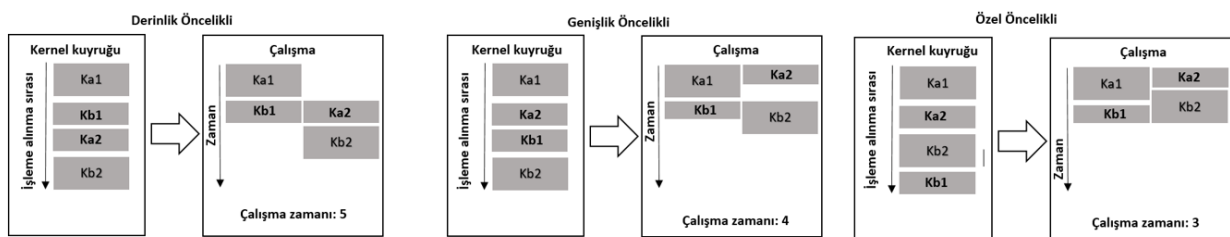
Programın CUDA işlemlerini işleme alma sırasına göre farklı stream'lerdeki CUDA kernel'lerinin eş zamanlı çalışmasındaki süreç değişmektedir. Programımız hem derinlik öncelikli (depth first) hem de genişlik öncelikli (breadth first) yaklaşımla işlemleri sırayla ele alsın:



Hesaplama motoru kuyruğunda bir kernel'in çalıştırılmak üzere donanıma gönderilebilmesi için aynı stream'deki kendinden önceki kernel'lerin çalışmasının tamamlanması gerektiğini, farklı stream'lerde kendinden önce kuyruğa giren kernel'ler var ise o kernel'lerin çalıştırılmak üzere donanıma gönderilmiş olması gerektiğini ve donanımda yeterince kaynağın var olması gerektiğini öğrenmiştik. Bu örnekte Kb1'in çalıştırılması için kendinden önce kuyruğa gelen ve kendisiyle aynı stream'de olan Ka1'in çalıştırılmasının tamamlanması beklenmektedir. Aynı şekilde Kb2'nin çalıştırılması için kendinden önce kuyruğa gelen ve kendisiyle aynı stream'de olan Ka2'nin çalıştırılmasının tamamlanması beklenmektedir. Bu iki durum hem derinlik öncelikli yaklaşımda hem de genişlik öncelikli yaklaşımda aynı etkiye sahiptir. Fakat Ka2'nin çalıştırılmasında farklılık oluşmaktadır. Eğer derinlik öncelikli yaklaşımı benimsersek Ka2 çalıştırılmak için Kb1'den sonra işleme alındığı için Kb1'in çalıştırılmak üzere donanıma gönderilmesini beklemek zorundadır. Dolayısıyla Ka1'in çalıştırılmasının tamamlanmasını beklemek zorunda kalmaktadır. Halbuki genişlik öncelikli yaklaşımı benimsersek programın Ka2'yi Kb1'den önce işleme almasını sağlayıp Ka2'nin Ka1 ile eş zamanlı çalışmasını sağlayabiliriz. Böylelikle çalışma zamanını 3'ten 2'ye düşürmüş oluruz. Bu arada her bir kernel SM kaynaklarının yarısını kullandığı için aynı anda iki farklı kernel'i çalıştırabilmiş olmaktadır. Şimdiki örnekte ise yine iki tane stream oluşturmuş olalım ve sadece CUDA kernel işlemleri olsun. Bu sefer kernel'lerin çalışma süresi farklı büyüklüklerde olsun. Aşağıda kernel'ler ile ilgili bilgiler verilmektedir:

- Stream 1: Ka1 {2 birim}, Kb1 {1 birim}
- Stream 2: Ka2 {1 birim}, Kb2 {2 birim}
- Her bir kernel SM kaynaklarının yarısını kullansın.

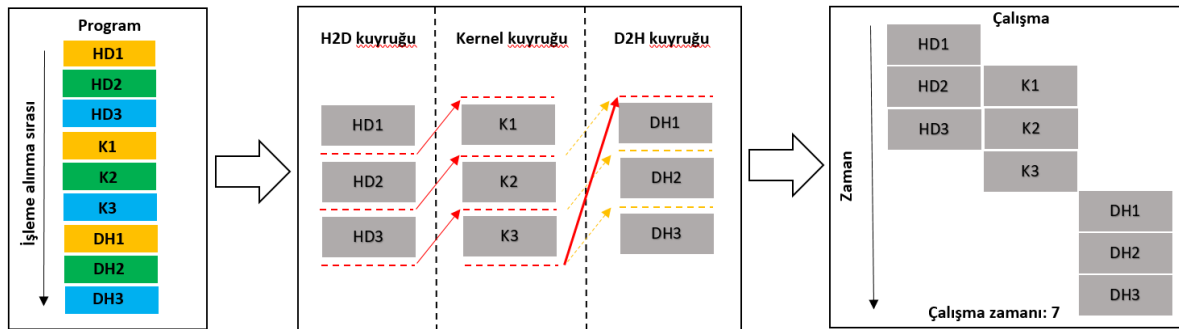
Programın CUDA işlemlerini işleme alma sırasına göre farklı stream'lerdeki CUDA kernel'lerinin eş zamanlı çalışmasındaki süreç değişmektedir. Programımız hem derinlik öncelikli (depth first) hem genişlik öncelikli (breadth first) hem de özel öncelikli yaklaşımlarla işlemleri sırayla ele alsın:



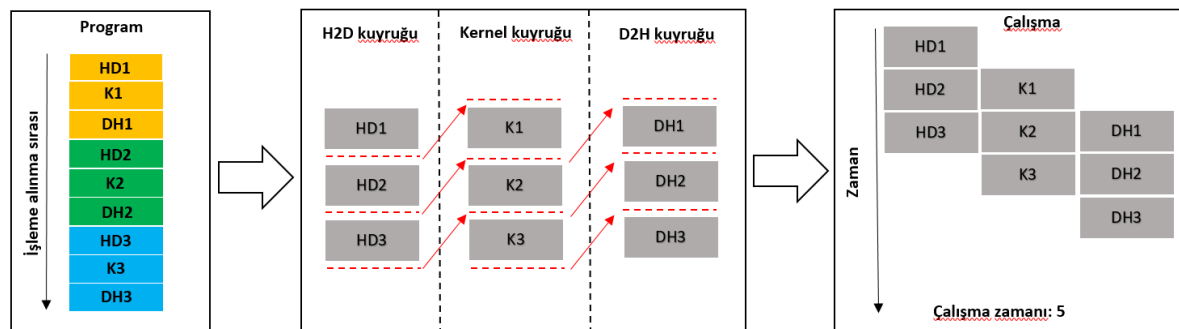
Program işlemleri derinlik öncelik ile ele aldığı anda önceki örnekte olduğu gibi stream 2'deki CUDA kernel'ler çalıştırılmak üzere donanıma gönderilebilmeleri için stream 1'deki kernel'lerin çalıştırılmak

üzere donanıma gönderilmesini beklemektedirler. Bu durumun programın farklı stream'lerdeki kernel'leri eş zamanlı çalıştırmasında sıkıntı yarattığını görmüştük. Bu durumu ortadan kaldırmak için Kb1 ile Ka2 kernel'lerinin işleme alınma sırasını değiştirmiştik. Fakat bu örnekte Kb2 Kb1'in çalıştırılmak üzere donanıma gönderilmesini beklemek zorunda olduğu için Ka2 bitir bitmez Kb2'yi çalıştıramıyoruz. Ka2'nin çalışma süresi Ka1 ve Kb2'nin yarısı olduğundan dolayı Ka2'nin çalıştırılmasının bitmesiyle Kb2'nin çalıştırılmaya başlama süresi arasında bir gecikme olmaktadır. Bu durumu da ortadan kaldırmak için özel bir öncelik tanımladık ve Kb1 ile Kb2 kernel'lerinin işleme alınma sıralarını değiştirdik. Böylelikle Ka2'nin çalıştırılması bitir bitir bitmek Kb2 çalıştırılmaya başlanmakta ve iki stream'e ait kernel'ler tam olarak eş zamanlı çalışmış olmaktadır.

Eş zamanlı kernel çizelgelemede özel bir duruma dikkat etmek gerekir. Normalde aynı stream'deki bir işlem tamamlandığı zaman diğer işleme geçildiğinde daha önce öğrendiğimiz gibi kuyruklar arası bir sinyal gönderilir. Fakat hesaplama motoru kuyruğundaki farklı stream'lere ait kernel'ler eş zamanlı çalıştırılırken bu kernel'lerin program tarafından işleme alınma sıraları ardışık sırada ise en son çalıştırılan kernel'in çalışması tamamlanana kadar bu sinyaller ertelenmektedir. Bu yüzden bazen bu erteleme diğer kuyrukları bloklayabilir. Bu durumu gösteren bir örnek için üç tane stream oluşturalım. Her bir stream sırasıyla bir tane CPU'dan GPU'ya veri transferi (HD), bir tane CUDA kernel (K) çalıştırma ve bir tane GPU'dan CPU'ya veri transferi (DH) işlemi yapsın. Genişlik öncelikli yaklaşımı kullanarak programımızı yazarsak aşağıdaki gibi senaryo gerçekleşecektir:



Görüldüğü gibi farklı stream'lerdeki kernel'ler programda ardışık olarak işleme alındığı için kernel kuyruğundaki K1'in çalıştırılması bittikten sonra D2H kuyruğuna gitmesi gereken sinyal ertelenmektedir. Aynı şekilde K2'nin çalıştırılması bittikten sonra D2H kuyruğuna gitmesi gereken sinyal de ertelenmektedir. Bu yüzden DH1 ve DH2 işlemleri çalıştırılmak için K3'ün çalıştırılmasının tamamlanmasını beklemektedirler. Bu sinyal erteleme yüzünden D2H kuyruğundaki işlemlerin çalıştırılması engellenmektedir. Bu problemi çözmek için derinlik öncelikli yaklaşımı benimserseniz aşağıdaki gibi bir senaryo gerçekleşecektir:



Bu senaryoda farklı stream'lerde çalışan kernel'ler program tarafından ardışık sırada işleme alınmadıklarından dolayı ilgili stream'deki kernel'in çalışması biter bitmez D2H kuyruğuna sinyal gönderilip ilgili veri transfer işlemi anında yapılmaya başlamaktadır. Böylelikle sinyalin ertelenmesi durumu ortadan kalkıp D2H kuyruğunun bloklanması ortadan kalkmıştır. Çalışma zamanı da 7'den 5'e düşmüştür.

Fermi mimarisinde stream'lerdeki işlemlerin işleme alınma sıralarına göre kuyruklara gönderildiğini öğrenmiştik. Bu işleme alınma sıraları yazılım tarafından kontrol edilmektedir. Kepler mimarisi ile sunulan Donanımsal İş Kuyruğu (Hardware Work Queue (HWQ)) yapısı ile stream'lerdeki işlemlerin planlanması CPU'ya geri dönmeyen donanım düzeyinde yapılması sağlanmıştır. Bu yapı ile Fermi mimarisinden farklı olarak her stream'in işlemleri o stream'e ait bir iş kuyruğunda işleme alınma sırasına konulmaktadır. Böylelikle farklı stream'lerdeki işlemler birbirlerini beklemek zorunda kalmadan eş zamanlı olarak çalışma imkanı bulmaktadır. Fakat farklı stream'lerdeki CUDA kernel'ler eş zamanlı çalışabilirken veri transferi işlemlerinde tek bir Kopyalama Motoru Kuyruğu olduğundan dolayı aynı yönde tek bir veri transferi kısıtlaması devam etmektedir. Pascal mimarisinde de bu kısıtlama devam ederken Volta mimarisinde çift yönlü üçüncü bir Kopyalama Motoru eklenmiştir. Böylelikle yönlerden herhangi birinde aynı anda iki tane veri transferi yapılabilir. Ampere mimarisinde de çift yönlü üçüncü bir Kopyalama Motoru eklenmiştir. Bu mimarinin bazı modellerinde ise Kopyalama Motoru sayısı 6'ya kadar çıkmaktadır. Hopper mimarisi ile Kopyalama Motorları önceki mimarilere göre daha da geliştirilmiştir. Modeline göre 6'dan fazla Kopyalama Motoruna sahip olabilen bu mimari hem farklı hem de aynı yönlü veri transferi işlemlerini farklı stream'ler üzerinden donanım düzeyinde eş zamanlı yürütebilen en gelişmiş CUDA mimarisidir. Hesaplama Motoru Kuyruğu sayısı ise Fermi'den sonraki mimarilerde aynı kalmış olsa da bu kuyruk giderek daha akıllı, daha verimli, daha paralel bir şekilde çalışan bir yapı haline getirilmiştir. Özellikle iş planlama ve donanımsal eşzamanlılık yönleri geliştirilmiştir.

Kepler mimarisindeki HWQ yapısı ile sunulan temelde donanımın CPU müdahalesi olmadan işleri sıraya koyup işleyebilmesi kavramı sonraki mimarilerde korunmakla beraber hem mimari olarak genişletilmiş hem de işlevsel olarak geliştirilmiştir. Özellikle Volta, Ampere ve Hopper mimarileri ile birlikte HWQ yapısı stream'deki işlemlerin planlanmasının ötesinde CUDA Graph, asenkron işlemler, doğrudan SM'den HWQ'ya erişim gibi daha kapsamlı donanım özelliklerinin temeli haline gelmiştir.