TÜBİTAK ULAKBİM

TRUBA Turkish Science e-Infrastructure

CUDA Programlamaya Giriş

EURO

Dr. Özcan Dülger
Bilgisayar Mühendisliği, Orta Doğu Teknik Üniversitesi
Bilgisayar Mühendisliği, Artvin Çoruh Üniversitesi

ORTA DOĞU TEKNİK ÜNİVERSİTESİ MIDDLE EAST TECHNICAL UNIVERSITY

HPC HIGH PERFORMANCE COMPUTING HPC@ODTÜ

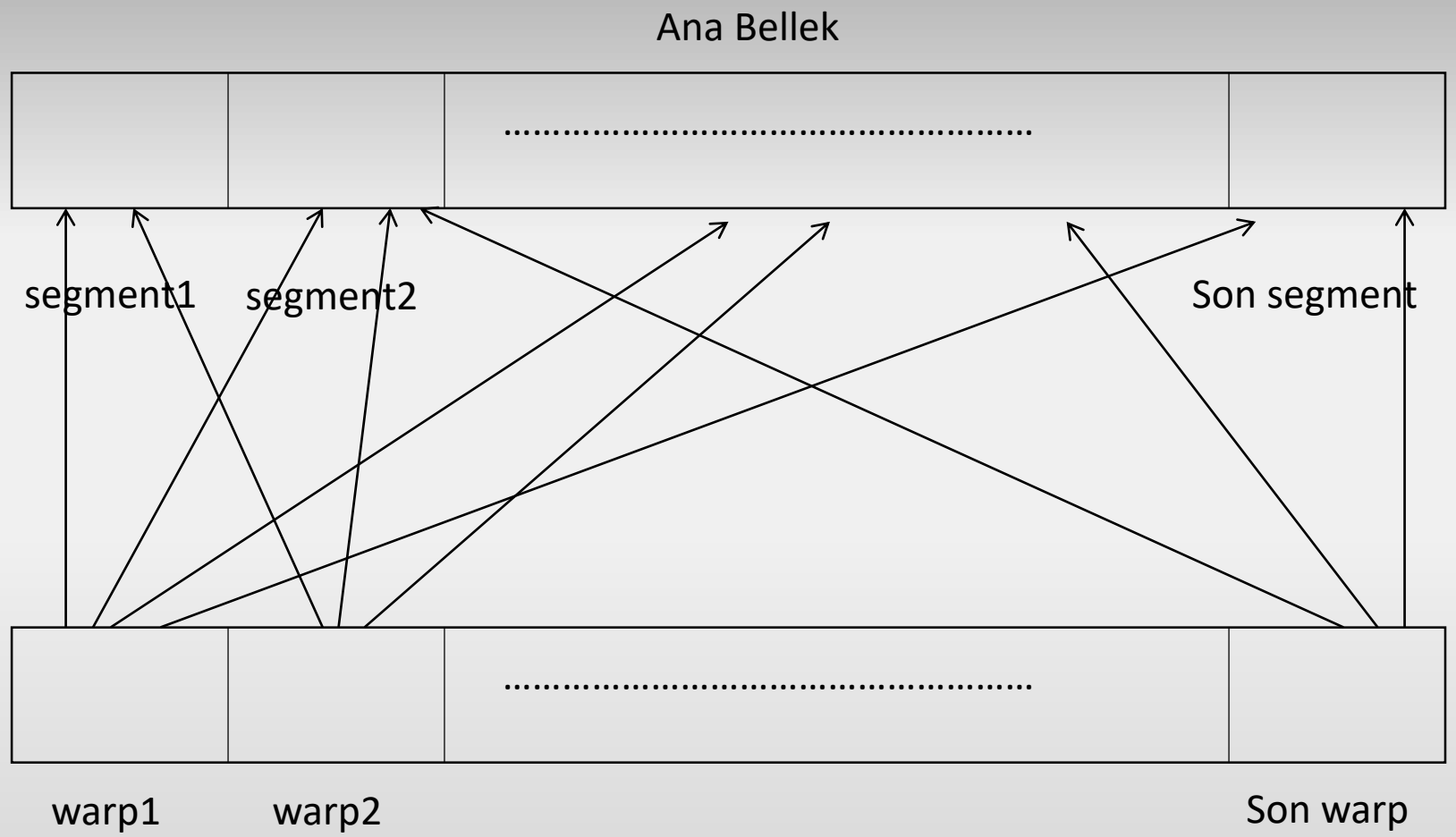
Eğitim İçeriği

- 1.Gün:
  - CPU mimarisi ile GPU mimarisinin karşılaştırılması
  - CUDA Programlamaya Giriş
  - Lab Oturumu-1
    - Vector Addition
- **2.Gün:**
  - **Ana Belleğe Düzenli Erişim (Coalesced Access to Global Memory) ve Warp Iraksaklığı (Warp Divergence)**
  - **CUDA Streams ve Çoklu-GPU (Multi-GPU)**
  - **Lab Oturumu-2**
    - **Vector Addition with Streams**

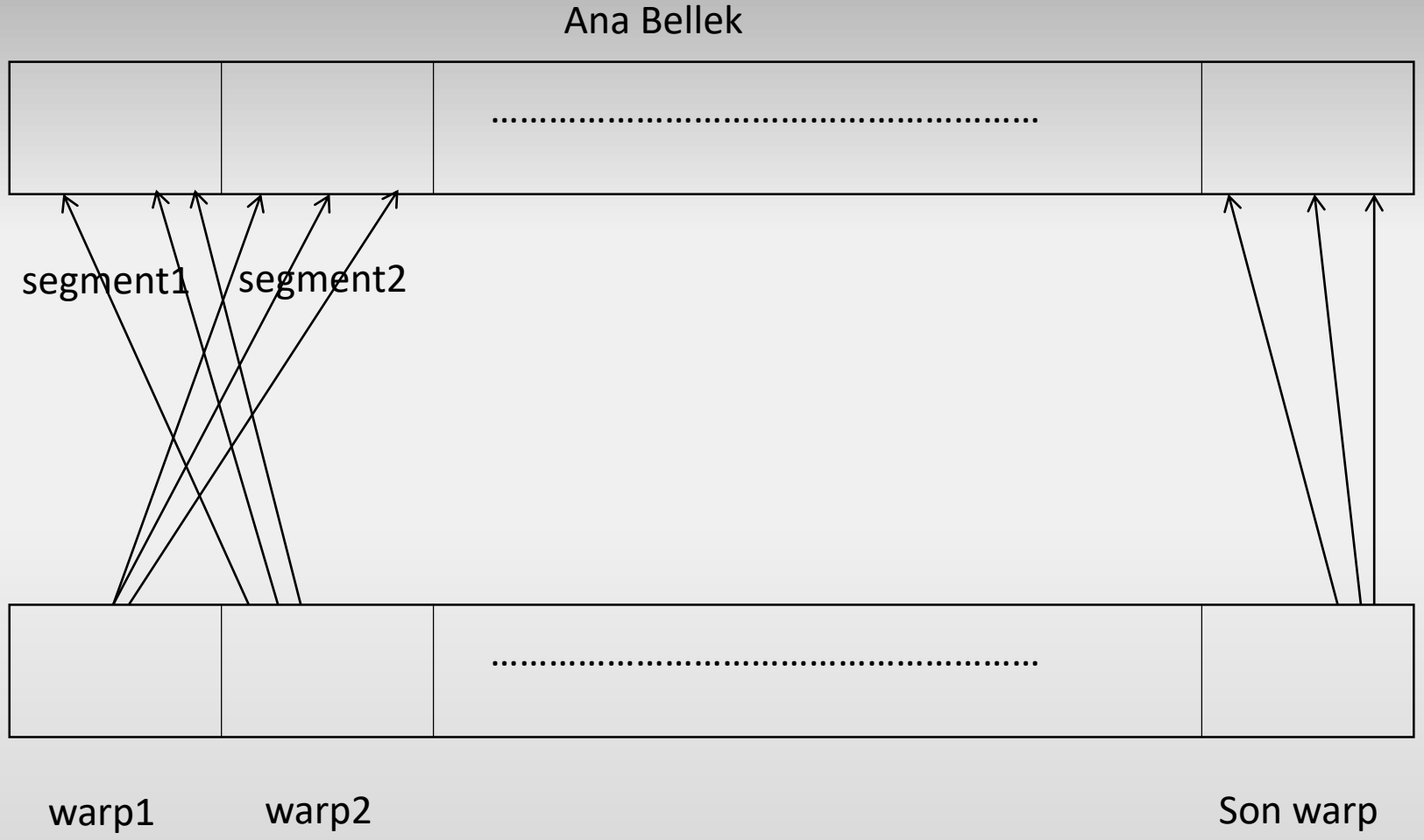
Ana Belleğe Düzenli Erişim (Coalesced Access to Global Memory)

- Ana bellek üzerindeki yazma ve okuma işlemleri segment'ler halinde olmaktadır
- Warp içindeki thread'ler birbirleri ile fiziksel olarak bağlıdırlar. Yani bir warp'ın bir komutu çalıştırmayı bitirmesi için o warp içindeki tüm thread'lerin o komutu çalıştırmayı bitirmesi lazım
- Ana bellek işlemlerinde, warp içindeki thread'ler ana belleğin farklı segmentlerine erişiyor ise o işlemlerin gerçekleştirilmesi serileşmektedir

Dağınık Erişim



Düzenli Erişim



```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    for(int counter = 0;counter < data_size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions in a period for 'data_size'

    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    curandInitializer RNGs(data_size);//Creating a generator for 'non_coalesced_access' kernel

    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kerne
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run
}
```

Load işlemi bir transaction sürüyor

Her bir thread için rastgele üreticilerin durum değişkenleri ana bellekten düzenli şekilde okunuyor

Load işlemi en fazla 32 transaction sürüyor

Not: Kernel çalışma zamanlarını ölçmek için cudaEventRecord komutunu kullandık

XORWOW Generator

EURO

```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 32768;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
```

Metrics:

gld_transactions: Number of global memory load transactions

gld_transactions_per_request: Average number of global memory load transactions performed for each global memory load

```
Exec. Time of 'full_coalesced_access' kernel = 0.000113152
Exec. Time of 'non_coalesced_access' kernel = 0.00168758
Speed Up = 14.9143X
```

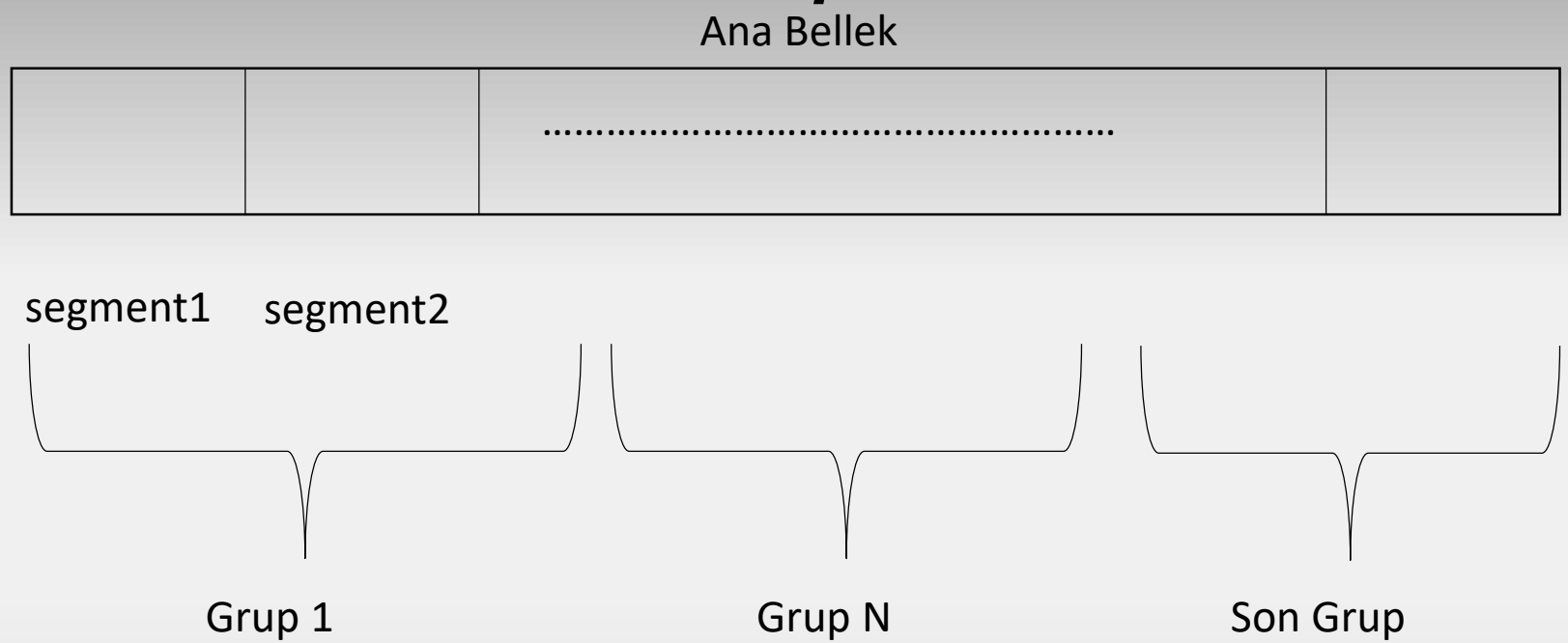
```
==1430== Profiling result:
==1430== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                         gld_transactions                   Global Load Transactions      204800      204800      204800
          1               gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                         gld_transactions                   Global Load Transactions     6461956     6461956     6461956
          1               gld_transactions_per_request      Global Load Transactions Per Request   30.633514   30.633514   30.633514
```

```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.00989446
Exec. Time of 'non_coalesced_access' kernel = 0.516703
Speed Up = 52.2214X
```

```
==1569== Profiling result:
==1569== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                         gld_transactions                   Global Load Transactions    26214400    26214400    26214400
          1               gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                         gld_transactions                   Global Load Transactions   839546946   839546946   839546946
          1               gld_transactions_per_request      Global Load Transactions Per Request   31.093373   31.093373   31.093373
```

Gruplama

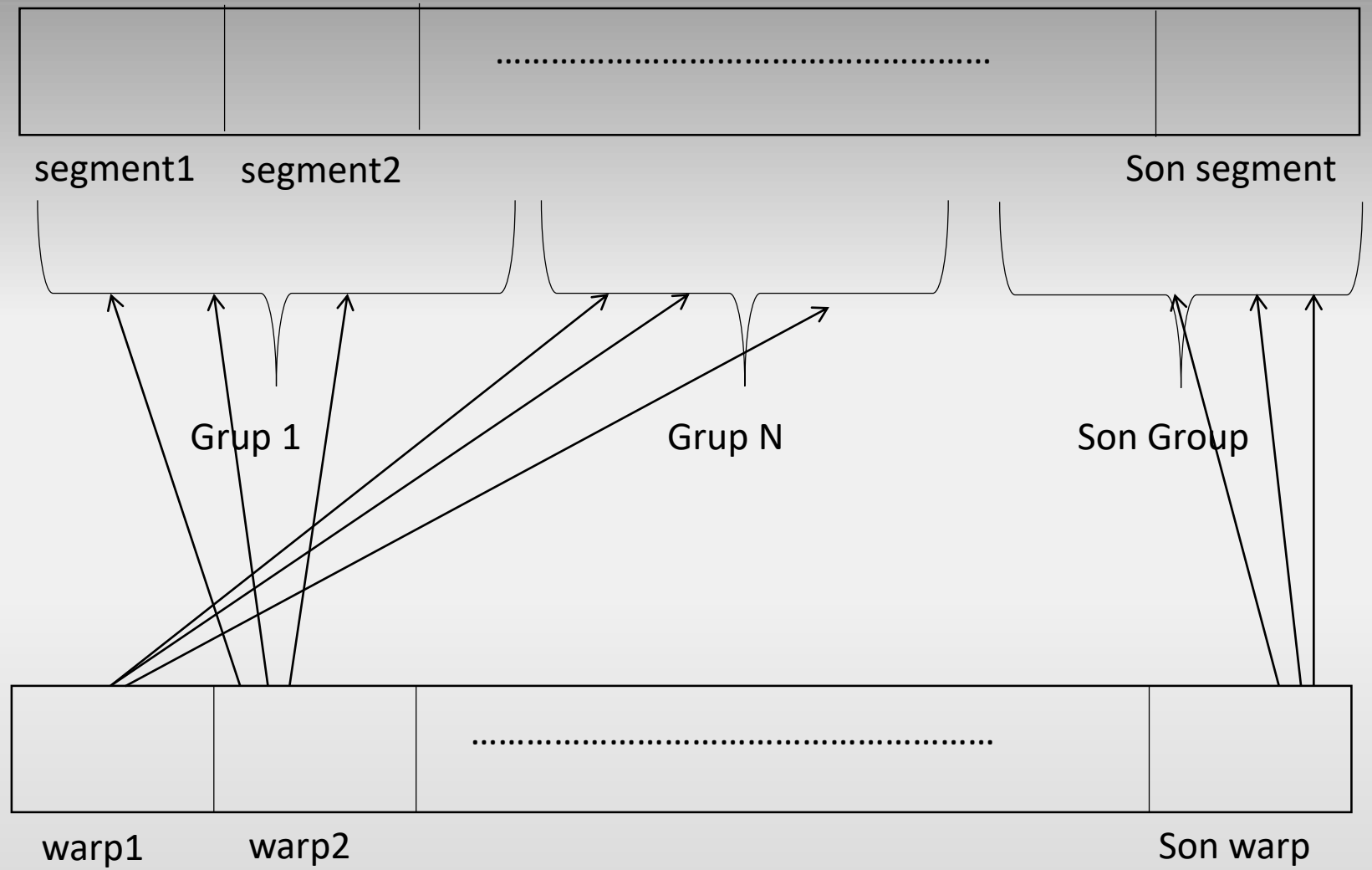


- Bir grup birbirini takip eden segment'lerden oluşuyor
- Bir gruptaki segment sayısı:
  - 1 ve data_size/32 arasında (32 segmentteki veri sayısı olmakta)

Ref: Dülger, Ö., Oğuztüzün, H. & Demirekler, M. Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit. J Sign Process Syst 90, 433–447 (2018)

Gruplama

Ana Bellek



```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
GS)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state1,state2;// States of the generators
    RNGs1.load(state1,tid);// Loading the state of first generator
    RNGs2.load(state2,tid);// Loading the state of second generator

    unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    for(int counter = 0;counter < data_size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions for 'data_size'

    unsigned int segment_size = 128;//Number of bytes of a segment
    unsigned int group_size = 16*(segment_size/4);//Number of data in a group
    unsigned int group_count = data_size/group_size;//Number of groups for 'data_size'
    unsigned int NP_group_count = (unsigned long int)pow(2,32)/group_count;// Number of partitions for 'group_count'
    unsigned int NP_group_size = (unsigned long int)pow(2,32)/group_size;// Number of partitions for 'group_size

    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kernel
    cudaDeviceSynchronize();//Waits until the kernel completes its run
}
```

- Bir gruptaki segment sayısı 16'dır
- Ana bellek işlemleri en çok 16 transaction'da tamamlanmakta

```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
GS)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state1,state2;// States of the generators
    RNGs1.load(state1,tid);// Loading the state of first generator
    RNGs2.load(state2,tid);// Loading the state of second generator

    unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 32768;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.000113216
Exec. Time of 'semi_coalesced_access' kernel = 0.000788544
Speed Up = 6.96495X
Exec. Time of 'non_coalesced_access' kernel = 0.00168253
Speed Up = 14.8612X
```

```
==1748== Profiling result:
==1748== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
       Kernel: full_coalesced_access(float*, float*, float*)
         1                            gld_transactions                   Global Load Transactions      204800      204800      204800
         1                gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
       Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
         1                            gld_transactions                   Global Load Transactions     2870920     2870920     2870920
         1                gld_transactions_per_request      Global Load Transactions Per Request   13.414511   13.414511   13.414511
       Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
         1                            gld_transactions                   Global Load Transactions     6461782     6461782     6461782
         1                gld_transactions_per_request      Global Load Transactions Per Request   30.632689   30.632689   30.632689
```

```
1 __global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
  GS)
2 {
3     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
4
5     curandState_t state1,state2;// States of the generators
6     RNGs1.load(state1,tid);// Loading the state of first generator
7     RNGs2.load(state2,tid);// Loading the state of second generator
8
9     unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
10    unsigned int index,i;
11
12    for(i=0;i<100;i++)
13    {
14        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
15        C[tid] = A[index] + B[index];//Vector addition
16    }
17 }
18
19 int main(int argc, char **argv)
20 {
21    unsigned int data_size = 4194304;//Data size
22    float *A_host,*B_host,*C_host ;//Host Arrays
23    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.00996032
Exec. Time of 'semi_coalesced_access' kernel = 0.221406
Speed Up = 22.2288X
Exec. Time of 'non_coalesced_access' kernel = 0.516618
Speed Up = 51.8676X
```

```
==2090== Profiling result:
==2090== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
       Kernel: full_coalesced_access(float*, float*, float*)
         1                            gld_transactions                   Global Load Transactions    26214400    26214400    26214400
         1                gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
       Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
         1                            gld_transactions                   Global Load Transactions   367412642   367412642   367412642
         1                gld_transactions_per_request      Global Load Transactions Per Request   13.412134   13.412134   13.412134
       Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
         1                            gld_transactions                   Global Load Transactions   839548386   839548386   839548386
         1                gld_transactions_per_request      Global Load Transactions Per Request   31.093427   31.093427   31.093427
```

Warp Iraksaklığı (Warp Divergence)

- ‘If-Else’ gibi yapıları bir warp tek bir komut olarak çalıştırmaktadır
- Bir warp’ın ‘If-Else’ komutunu çalıştırmayı bitirebilmesi için o warp’taki tüm thread’lerin bu ‘If-Else’ komutunu çalıştırmayı bitirmesi gerekir
- Eğer warp içindeki thread’ler ‘If-Else’ komutunun değişik koşullarındaki komutları çalıştırıyor ise bu koşullardaki komutların çalıştırılması serileşmektedir
- if(tid %2 == 0)//tid is global thread id

  ……..

  else

  ……..
- Önce warp içindeki global thread id’si çift sayı olan thread’ler “if” koşulunu gerçekleştirmekte ve kalan thread’ler beklemekte
- Sonra warp içindeki global thread id’si tek sayı olan thread’ler “else” koşulunu gerçekleştirmekte ve kalan thread’ler beklemekte

Warp Divergence

- Warp içindeki thread'lerin 'if-Else' yapısının aynı koşulunu çalıştırması performans açısından oldukça önemlidir
- Thread id yerine warp id kullanarak bu durumu sağlayabiliriz
- if((tid/32) %2 == 0)//tid is the global thread id

  ……..

  else

  ……..
- Warp id'si çift olan thread'ler 'if' koşulunu çalıştırmakta
- Warp id'si tek olan thread'ler 'else' koşulunu çalıştırmakta
- Böylelikle "if-Else" yapısını çalıştırırken serileşme ortadan kalktı

```cpp
__global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( (tid/32) % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( (tid/32) % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( (tid/32) % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( (tid/32) % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

__global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( tid % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( tid % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( tid % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( tid % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    warp_no_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_no_divergence' kernel
    warp_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_divergence' kernel
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run
}
```

- Koşullar warp id'ye göre dağıtılmakta
- İlk warp toplama, ikinci warp çıkarma, üçüncü warp çarpma, dördüncü warp bölme, ……., işlemi yapmakta

- Koşullar thread id'ye göre dağıtılmakta
- İlk thread toplama, ikinci thread çıkarma, üçüncü thread çarpma, dördüncü thread bölme, ……., işlemi yapmakta

- Yeterince koşul yer almakta (4)
- Süre ölçümü için komut yeterince kez çalıştırılmakta (100)
- Bellek işlemleri düzenli, dolayısıyla warp divergence toplam süreye etki etmekte
- Kernel çalışma zamanını ölçmek için **cudaEventRecord** komutu kullanılmakta

```cuda
__global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( (tid/32) % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( (tid/32) % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( (tid/32) % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( (tid/32) % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

__global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( tid % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( tid % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( tid % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( tid % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

```
Exec. Time of 'warp_no_divergence' kernel = 0.0124316
Exec. Time of 'warp_divergence' kernel = 0.0385476
Speed Up = 3.10078X
```

```
==23306== Profiling result:
==23306== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: warp_divergence(float*, float*, float*)
          1                  warp_execution_efficiency              Warp Execution Efficiency      34.80%      34.80%      34.80%
        Kernel: warp_no_divergence(float*, float*, float*)
          1                  warp_execution_efficiency              Warp Execution Efficiency     100.00%     100.00%     100.00%
```

Metric:

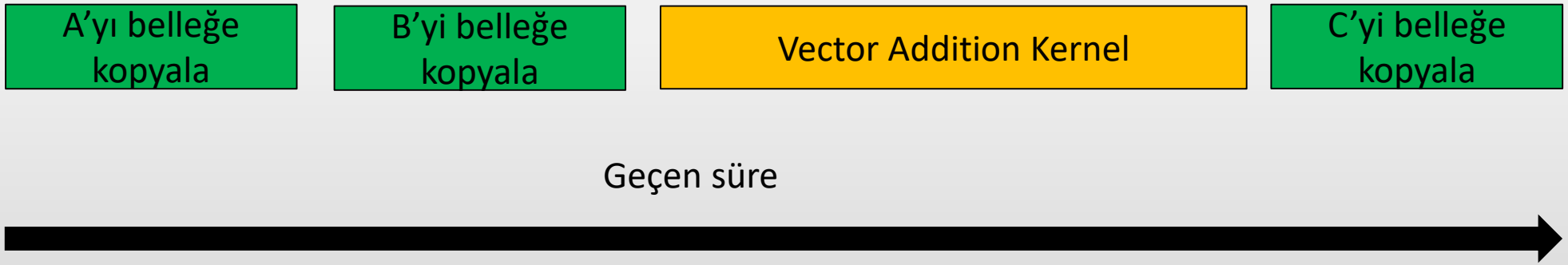
warp_execution_efficiency: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage

CUDA Streams

- “Task parallelism” farklı görevlerin (tasks) aynı veri üzerinde eş zamanlı olarak birden çok işlemcide çalışmasıyla ilgilenmektedir
- Aynı görevin verinin farklı kısımları üzerinde çalışmasıyla ise “data parallelism” ilgilenir
- CUDA’da “Task parallelism” CUDA stream ile sağlanabilmektedir
- CUDA stream’lerde aynı stream’deki işlemler seri olarak çalışır. (FIFO ilkesine göre)
- Bir stream’deki işlem başka bir stream’deki işlem ile eş zamanlı çalışabilir
  - Fakat bellekten veri transferlerinde herhangi bir yönde aynı anda tek bir transfer işlemi gerçekleşir
  - Aynı anda biri CPU’dan GPU’ya diğeri GPU’dan CPU’ya olmak üzere en fazla iki işlem gerçekleşir
- Her bir CUDA kodunda default bir stream (stream 0) oluşturulmaktadır. Bu stream’deki işlemler sonradan oluşturulan stream’lerdeki işlemler ile eş zamanlı çalışmamaktadır

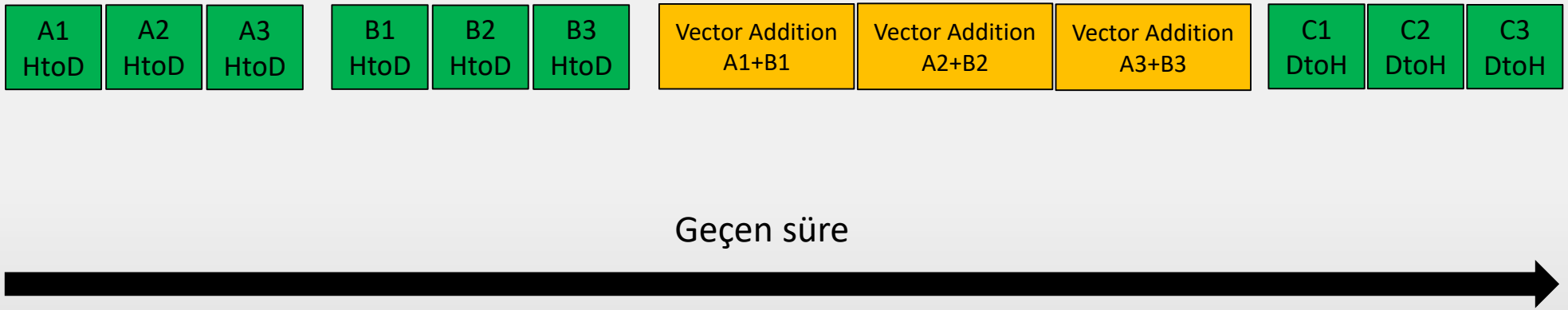
CUDA Streams

- Dünki eğitimdeki “vector addition” örneğini ele alalım:
  - cudaMemcpy ve CUDA kernel çalıştırma işlemleri seri olarak çalışmaktaydı
  - Bazen GPU boşta kalmakta, bazen veri aktarımının yapıldığı PCIe boşta kalmakta
  - “Task Parallelism” elde edebilmek pipelining tekniği uygulayacağız ve CUDA Stream kullanacağız



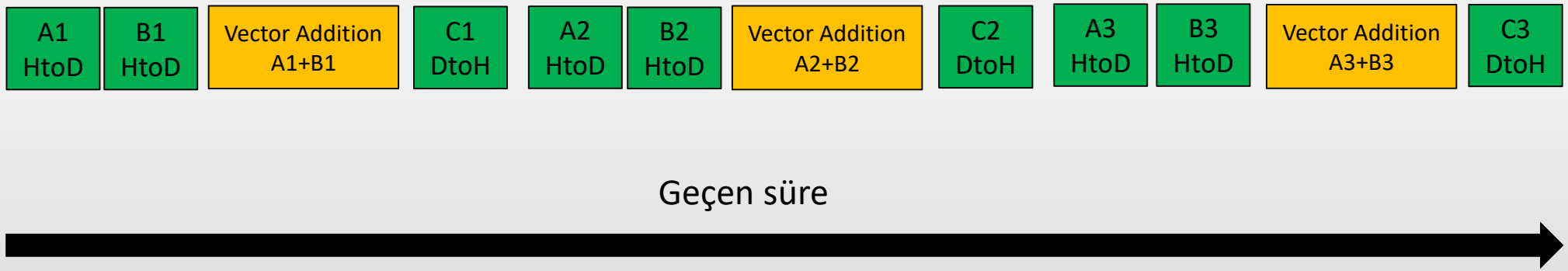
CUDA Streams ve Pipelining

- Veri setini küçük parçalara ayıracağız. Mesela üçe bölelim (A1,A2,A3 gibi)



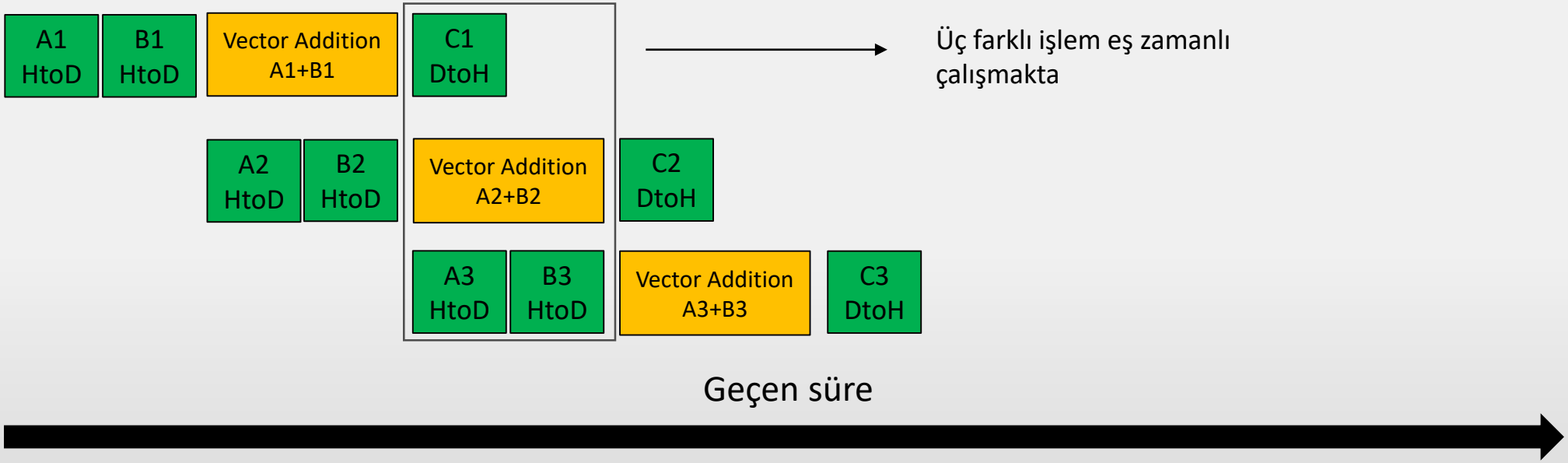
CUDA Streams ve Pipelining

- Bu işlemlerin sırasını düzenledikten sonra birbirinden bağımsız çalışan üç küçük “vector addition” işlemi elde ettik
- Böylelikle verilerin belleğe kopyalanma işlemlerini ve “vector addition” işlemlerini örtüştürebiliriz



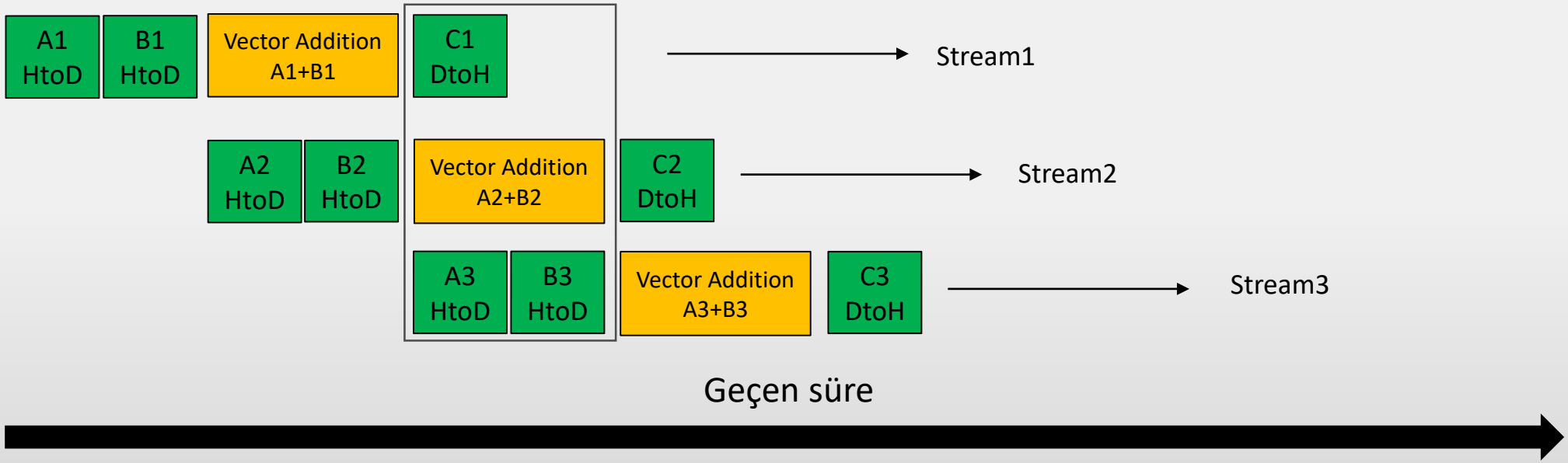
CUDA Streams ve Pipelining

- Örtüştürme tekniği için pipelining tekniğini kullanıyoruz
- Bu işlemleri eş zamanlı çalıştırmak için CUDA Stream'leri kullanacağız



CUDA Streams

- Her bir parça için bir stream oluşturabiliriz.
- Her bir stream'de sıraya alınmış işlemler (cudaMemcpyAsync, CUDA Kernel gibi) bulunmaktadır
- Farklı stream'lerdeki işlemler eş zamanlı çalışabilir (Task Parallelism)



CUDA Streams

```
int main()
{
    int *A_Host,*B_Host;//CPU'da dizi değişkenleri
    A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
    B_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor

    int *A_GPU,*B_GPU;//GPU'da dizi değişkenleri
    cudaMalloc(&A_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor
    cudaMalloc(&B_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor

    cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
    my_kernel<<<DimGrid,DimBlock>>>(A_GPU,B_GPU,size);//CUDA kernel çalıştırılıyor
    cudaMemcpy(B_Host,B_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı

    delete[] A_Host;//Dizi CPU belleğinden siliniyor
    delete[] B_Host;//Dizi CPU belleğinden siliniyor

    cudaFree(A_GPU);//Dizi GPU belleğinden siliniyor
    cudaFree(B_GPU);//Dizi GPU belleğinden siliniyor
}
```

```
int main()
{
    int *A_Host,*B_Host;//CPU'da dizi değişkenleri
    cudaMallocHost((void**)&A_Host, sizeof(int)*size);//CPU belleğinde yer açılıyor
    cudaMallocHost((void**)&B_Host, sizeof(int)*size);//CPU belleğinde yer açılıyor

    int *A1_GPU,*B1_GPU;//GPU'da stream1 için dizi değişkenleri
    int *A2_GPU,*B2_GPU;//GPU'da stream2 için dizi değişkenleri

    cudaMalloc(&A1_GPU,sizeof(int)*size/2);//GPU ana belleğinde yer açılıyor
    cudaMalloc(&B1_GPU,sizeof(int)*size/2);//GPU ana belleğinde yer açılıyor
    cudaMalloc(&A2_GPU,sizeof(int)*size/2);//GPU ana belleğinde yer açılıyor
    cudaMalloc(&B2_GPU,sizeof(int)*size/2);//GPU ana belleğinde yer açılıyor

    cudaStream_t stream[2];//Stream dizisi tanımlanıyor
    cudaStreamCreate(&stream[0]);//Stream1 yaratılıyor
    cudaStreamCreate(&stream[1]);//Stream2 yaratılıyor

    //Stream1 işlemleri başlıyor. İşlemler Host'a göre asenkronize çalışıyor
    cudaMemcpyAsync(A1_GPU,A_Host+0*(size/2),sizeof(int)*size/2,cudaMemcpyHostToDevice,stream[0]);//CPU'dan GPU'ya veri aktarımı
    my_kernel<<<DimGrid,DimBlock,0,stream[0]>>>(A1_GPU,B1_GPU,size/2);//CUDA kernel çalıştırılıyor
    cudaMemcpyAsync(B_Host+0*(size/2),B1_GPU,sizeof(int)*size/2,cudaMemcpyDeviceToHost,stream[0]);//GPU'dan CPU'ya veri aktarımı

    //Stream2 işlemleri başlıyor. İşlemler Host'a göre asenkronize çalışıyor
    cudaMemcpyAsync(A2_GPU,A_Host+1*(size/2),sizeof(int)*size/2,cudaMemcpyHostToDevice,stream[1]);//CPU'dan GPU'ya veri aktarımı
    my_kernel<<<DimGrid,DimBlock,0,stream[1]>>>(A2_GPU,B2_GPU,size/2);//CUDA kernel çalıştırılıyor
    cudaMemcpyAsync(B_Host+1*(size/2),B2_GPU,sizeof(int)*size/2,cudaMemcpyDeviceToHost,stream[1]);//GPU'dan CPU'ya veri aktarımı

    cudaStreamSynchronize(stream[0]);//Stream1'deki tüm işlemler bitene kadar program bekliyor
    cudaStreamSynchronize(stream[1]);//Stream2'deki tüm işlemler bitene kadar program bekliyor

    cudaStreamDestroy(stream[0]);//Stream1 yok ediliyor
    cudaStreamDestroy(stream[1]);//Stream2 yok ediliyor

    cudaFreeHost(A_Host);//Dizi CPU belleğinden siliniyor
    cudaFreeHost(B_Host);//Dizi CPU belleğinden siliniyor

    cudaFree(A1_GPU);//Dizi GPU belleğinden siliniyor
    cudaFree(B1_GPU);//Dizi GPU belleğinden siliniyor
    cudaFree(A2_GPU);//Dizi GPU belleğinden siliniyor
    cudaFree(B2_GPU);//Dizi GPU belleğinden siliniyor
}
```

Page-locked memory. Asenkronize ve daha hızlı veri transferi sağlıyor

Stream'ler yaratılıyor

3. parametre shared memory ile alakalı. Default değeri 0

Stream1 ve stream2'deki işlemler overlap edilmekte

Stream'ler senkronize ediliyor

Stream'ler yok ediliyor

Multi GPU

- Bir CPU birden çok GPU'ya eriştiğinde multi-GPU çözümler ortaya konulabilmektedir
- Her bir GPU'da ayrı ayrı CUDA komutları çalıştırılabilmektedir
- Bunun için cudaSetDevice(int GPU_ID) komutu kullanılmaktadır
  - Önce ilgili GPU bu komut ile aktif hale getirilir ve sonra ilgili CUDA komutları o GPU'da çalıştırılır
- İki ayrı stream kullanarak overlap ettiğimiz son örneği tek GPU'da overlap etmek yerine iki ayrı GPU'da eş zamanlı çalıştırmayı deneyelim

```
int main()
{
    int *A_Host,*B_Host;//CPU'da dizi değişkenleri
    cudaMallocHost((void**)&A_Host, sizeof(int)*size);//CPU belleğinde yer açılıyor
    cudaMallocHost((void**)&B_Host, sizeof(int)*size);//CPU belleğinde yer açılıyor

    int *A1_GPU,*B1_GPU;//GPU1 için dizi değişkenleri
    int *A2_GPU,*B2_GPU;//GPU2 için dizi değişkenleri

    cudaSetDevice(0);//1.GPU aktif hale getiriliyor
    cudaMalloc(&A1_GPU,sizeof(int)*size/2);//GPU1 ana belleğinde yer açılıyor
    cudaMalloc(&B1_GPU,sizeof(int)*size/2);//GPU1 ana belleğinde yer açılıyor
    cudaSetDevice(1);//2.GPU aktif hale getiriliyor
    cudaMalloc(&A2_GPU,sizeof(int)*size/2);//GPU2 ana belleğinde yer açılıyor
    cudaMalloc(&B2_GPU,sizeof(int)*size/2);//GPU2 ana belleğinde yer açılıyor

    cudaSetDevice(0);//1.GPU aktif hale getiriliyor
    //GPU1 işlemleri başlıyor. İşlemler Host'a göre asenkronize çalışıyor
    cudaMemcpyAsync(A1_GPU,A_Host+0*(size/2),sizeof(int)*size/2,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
    my_kernel<<<DimGrid,DimBlock>>>(A1_GPU,B1_GPU,size/2);//CUDA kernel çalıştırılıyor
    cudaMemcpyAsync(B_Host+0*(size/2),B1_GPU,sizeof(int)*size/2,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı

    cudaSetDevice(1);//2.GPU aktif hale getiriliyor
    //GPU2 işlemleri başlıyor. İşlemler Host'a göre asenkronize çalışıyor
    cudaMemcpyAsync(A2_GPU,A_Host+1*(size/2),sizeof(int)*size/2,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
    my_kernel<<<DimGrid,DimBlock>>>(A2_GPU,B2_GPU,size/2);//CUDA kernel çalıştırılıyor
    cudaMemcpyAsync(B_Host+1*(size/2),B2_GPU,sizeof(int)*size/2,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı

    cudaSetDevice(0);//1.GPU aktif hale getiriliyor
    cudaDeviceSynchronize();//GPU1'deki tüm işlemler bitene kadar program bekliyor
    cudaSetDevice(1);//2.GPU aktif hale getiriliyor
    cudaDeviceSynchronize();//GPU2'deki tüm işlemler bitene kadar program bekliyor

    cudaFreeHost(A_Host);//Dizi CPU belleğinden siliniyor
    cudaFreeHost(B_Host);//Dizi CPU belleğinden siliniyor

    cudaSetDevice(0);//1.GPU aktif hale getiriliyor
    cudaFree(A1_GPU);//Dizi GPU1 belleğinden siliniyor
    cudaFree(B1_GPU);//Dizi GPU1 belleğinden siliniyor
    cudaSetDevice(1);//2.GPU aktif hale getiriliyor
    cudaFree(A2_GPU);//Dizi GPU2 belleğinden siliniyor
    cudaFree(B2_GPU);//Dizi GPU2 belleğinden siliniyor
}
```

İlgili GPU aktif hale getiriliyor

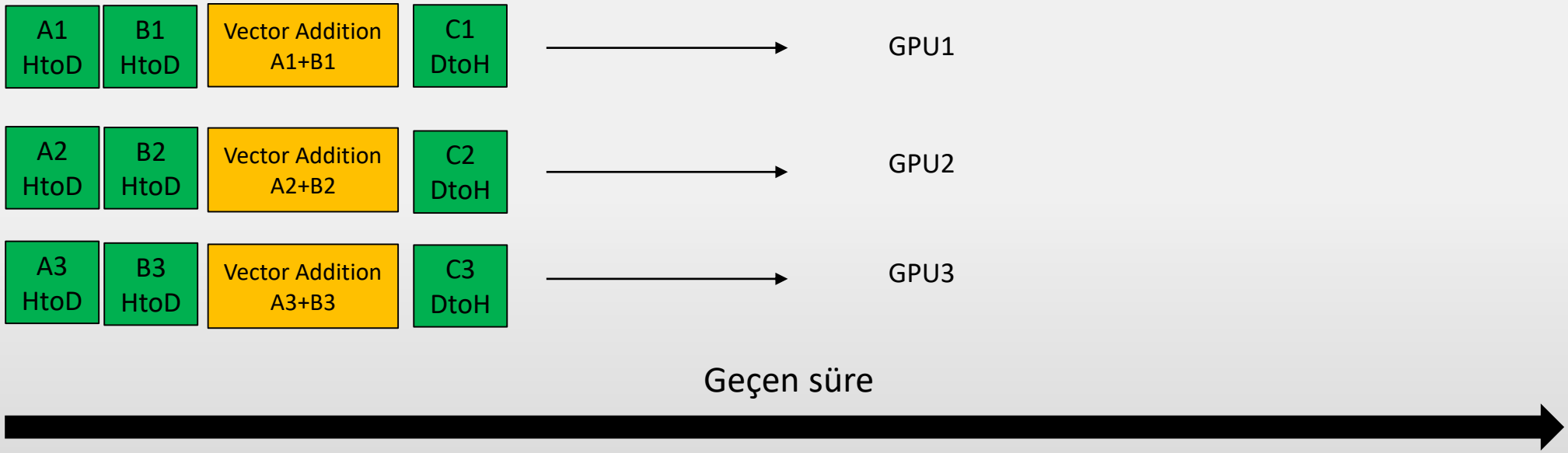
İlgili GPU aktif hale getiriliyor

GPU’lar eş zamanlı çalışabilmesi için işlemler asenkronize olmalıdır

EURO

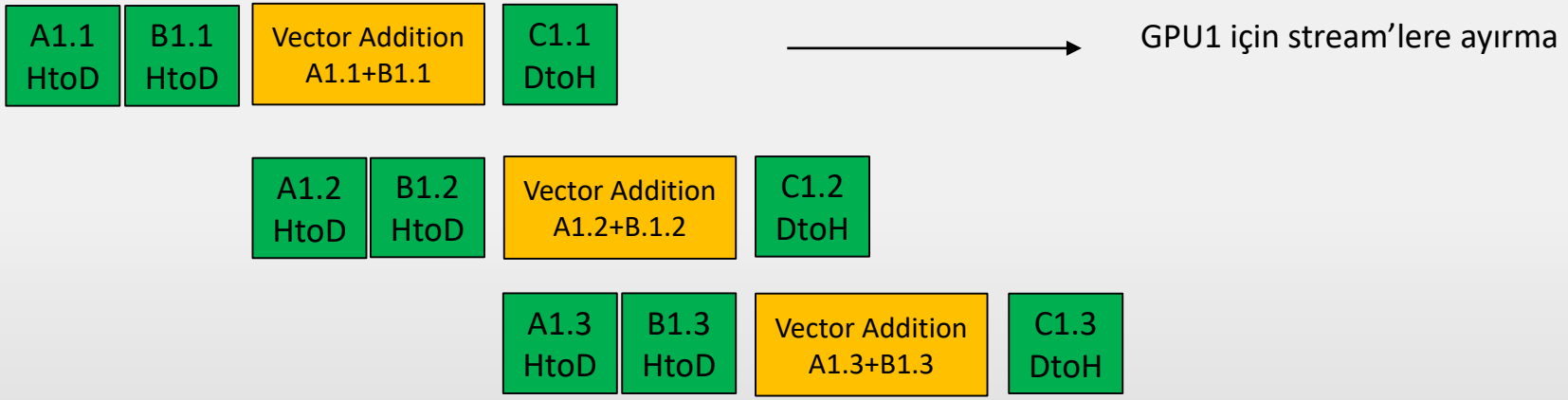
Multi GPU

- Tek GPU'da üç ayrı veri setine böldüğümüz vector addition örneğimizdeki her bir veri setini farklı bir GPU'da eş zamanlı çalıştırabiliriz



Multi GPU

- GPU'lara paylaştırdığımız verileri ayrıca her bir GPU içinde stream'lere ayırarak daha da hızlanma kazanabiliriz



Multi GPU

- GPU'lar aralarında direk bir veri aktarımı yapabilir (P2P – Peer to Peer)
  - Daha verimli bir iletişim olmakta ve CPU'nun belleğine gitmeden hızlı veri transferi sağlanmakta
- İlk olarak iki GPU arasında P2P aktif hale getirilmelidir
  - cudaDeviceEnablePeerAccess (int peerDevice, 0) -> P2P aktif hale geliyor
  - cudaDeviceDisablePeerAccess (int peerDevice) -> P2P pasif hale geliyor
  - cudaDeviceCanAccessPeer (int* canAccessPeer, int device, int peerDevice) -> P2P var mı kontrolü (0-yok, 1-var)
- cudaMemcpyPeer (void* dst, int dstDevice, const void* src, int srcDevice, size_t count)
  - İki GPU arasında P2P veri transferi için kullanılmakta

Teşekkürler!

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro

LAB OTURUMU-2

- Konu: Vector Addition with Streams
- Dosyalara Erişim: https://indico.truba.gov.tr/event/89/
- Yardımcı Eğitmenler:
  - Abdullah Doğan - ODTÜ Bilgisayar Mühendisliği (Room:1)
  - Alper Karamanlıoğlu - ODTÜ Bilgisayar Mühendisliği (Room:2)
  - Kadir Cenk Alpay - ODTÜ Bilgisayar Mühendisliği (Room:3)
  - Merve Taplı - ODTÜ Bilgisayar Mühendisliği (Room:3)
  - Ali Ata Adam - ODTÜ Havacılık ve Uzay Mühendisliği (Room:4)
  - Saeideh Nazirzadeh – ODTÜ Mühendislik Bilimleri (Zoom alt yapısı destek)