PERFORMANS OPTİMİZASYONU VE VERİMLİLİK

CUDA programlamada uygulama performansı; global belleğe yapılan erişimlerin coalesced yapıda gerçekleştirilmesi, warp seviyesinde kontrol akışı ayrışmalarının (warp divergence) minimize edilmesi ve GPU üzerindeki çoklu işlemcilerin (Streaming Multiprocessors – SMX) etkin biçimde kullanılması gibi temel etmenlere bağlıdır. Device occupancy ve SM efficiency gibi metrikler, GPU üzerindeki donanımsal kaynakların ne ölçüde etkin kullanıldığını nicel olarak ortaya koyarken; bellek erişim düzenleri ve paralel yürütme yapılarının uygulama performansı üzerindeki etkilerinin değerlendirilmesine olanak sağlar. İlk olarak deneylerde kullanılan Tesla K40 ekran kartının bazı teknik özelliklerini inceleyelim. Aşağıdaki tabloda Tesla K40 mimarisi ile alakalı bazı donanımsal ve mimari parametreler verilmektedir:

Property	Value	Property	Value
Architecture	Kepler	Global Memory	11520 MB
Number of SMX	15	Shared Memory	49152 Byte
CUDA Core	2880	L2 Cache	1572864 Byte
Core Clock	745 MHz	Segment Size	128 Byte
Max. Thread / SMX	2048	Warp Size	32
Max. Thread / Block	1024	Max. Block / SMX	16

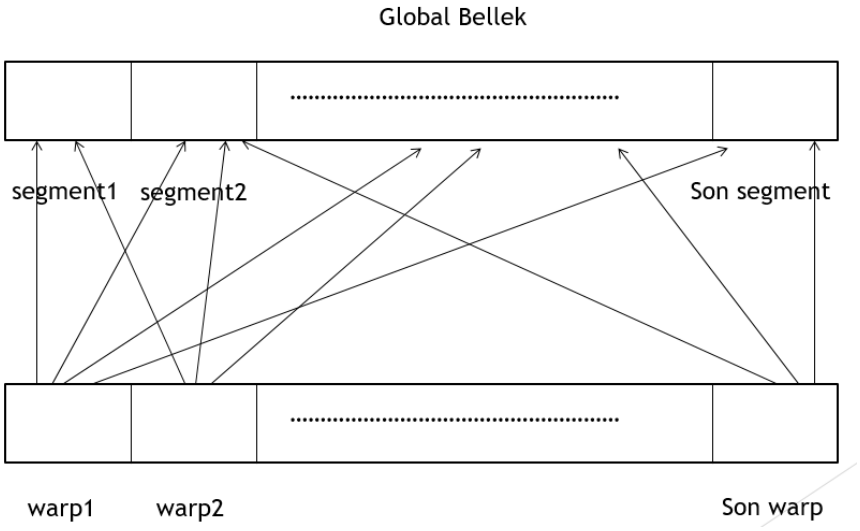
Bu mimari ile alakalı bazı parametrelere aşağıda maddeler halinde değinilmektedir:

- Bu mimaride 15 tane SMX bulunmaktadır ve her bir SMX 192 CUDA çekirdeğine sahiptir.
- Warp donanımsal bir kavram olup yazılım tarafında ona direkt bir müdahale mümkün değildir. Bir warp 32 tane thread'e sahiptir. Bu thread'ler birbirlerine fiziksel olarak bağlıdırlar. Yani bir warp'ın bir komutu çalıştırıp bitirebilmesi için o warp içindeki tüm thread'lerin aynı komutu çalıştırıp bitirmesi gerekir.
- Bir SMX'te aynı anda en fazla 2048 thread aktif olarak çalışabilmektedir. Yani en fazla 64 warp aktif olarak çalışabilmektedir.
- Warp'lar CUDA çekirdeklerine çalıştırılmak üzere gönderilir. Bir warp bir komutu çalıştırmayı bitirdikten sonra sırada bekleyen başka bir warp çekirdeklere çalıştırılmak üzere gönderilir.
- Bir blok en fazla 1024 thread'e sahip olabilir. Bu yazılımsal bir parametre olup değeri kodu yazarken belirlememiz gerekir. Bloklar SMX'lere çalıştırılmak üzere gönderilir. Bir SMX'te en fazla 16 blok aynı anda aktif olarak çalışabilir. Bu parametrenin bir arada değerlendirilmesi performans ve verimlilik açısından oldukça önemlidir.
- Global bellek üzerinde okuma/yazma işlemleri bellek segmentleri üzerinde gerçekleşmektedir. Yani bir warp'taki thread'ler aynı bellek segment üzerinde verilere erişiyorsa bu işlemler tek transaction'da tamamlanır. Diğer türlü bu işlemler birden çok transaction'da tamamlanır. Tesla K40 mimarisinde segment büyüklükleri 128 bayttır.

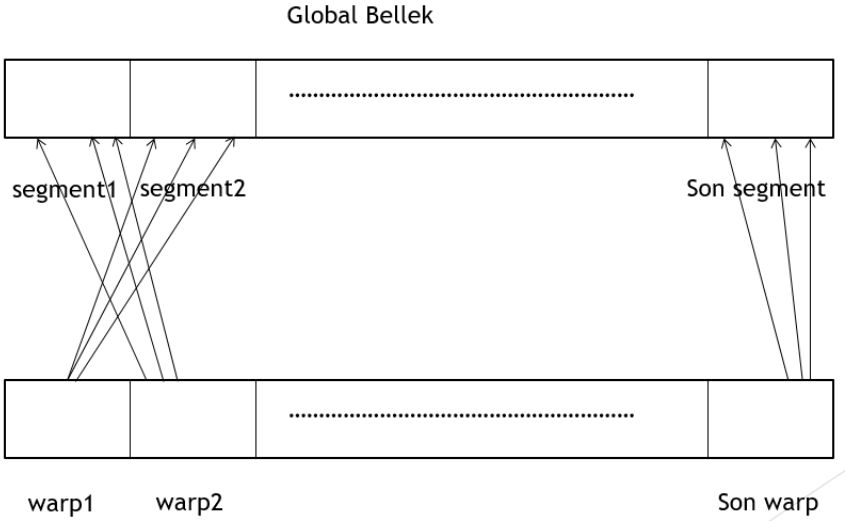
1. Global Belleğe Coalesced (Birleştirilmiş) Erişim:

GPU'nun global belleğine yapılan okuma/yazma işlemleri segmentler halinde gerçekleştirilmektedir. Aynı warp'taki thread'ler birbirlerine fiziksel olarak bağlıdırlar. Yani bir warp'ın bir komutu çalıştırıp bitirebilmesi için o warp'taki tüm thread'lerin aynı komutu çalıştırıp bitirmesi gerekir. Eğer warp içindeki tüm thread'ler aynı segment üzerinde bir işlem yapıyorsa bu işlem 1 transaction'da tamamlanmaktadır. Eğer thead'ler farklı segmentler üzerinde işlem yapıyorsa bu işlem

serileşmektedir. Aşağıdaki şekilde bir CUDA kernel için çalışan warp'lardaki thread'lerin global bellekteki farklı segmentlere dağınık erişimi gösterilmektedir:



Aynı warp'taki thread'lerin aynı segment üzerinde işlem yapmaya zorlar isek bellek işlemlerinin serileşmesinin önüne geçmiş oluruz. Aşağıdaki şekilde bir CUDA kernel için çalışan warp'lardaki thread'lerin global bellekteki aynı segmente birleştirilmiş (coalesced) erişimi gösterilmektedir:



Vektör toplama örneği üzerinden dağınık (non-coalesced) erişim ile birleştirilmiş (coalesced) erişimi hem çalışma zamanı açısından hem de transaction sayısı açısından inceleyelim. Aşağıda bu deneyi yapmak için kullandığımız kodun bir kısmı verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Memory Coalesced Access isimli kodu indirebilirsiniz. Bu kodda kullanılmış olan random number generator'ların kodlarını içeren "RNG_functions.cuh" isimli başlık dosyasına Yardımcı Dosyalar bölümünde verilen RNG Functions Header isimli dosyayı indirerek erişebilirsiniz.

```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
}
```

Okuma işlemi bir transaction sürüyor

Her bir thread generator'ın durumlarını bellekten coalesced şekilde yüklemektedir

Okuma işlemi 32 transaction sayısına kadar çıkabilir

Not: Kernel çalışma sürelerini ölçmek için cudaEventRecord kullanıyoruz.

XORWOW Generators

Kod ile ilgili önemli kısımlara maddeler halinde değinelim:

- full_coalesced_access kernel'inde A ve B değişkenlerinin işaret ettiği bellek bölgelerine tamamen birleştirilmiş (coalesced) bir şekilde erişim sağlanmaktadır. Bunu sağlamak için her bir thread'in kendine özgü olan global thread numarası (0 ile toplam thread sayısı-1 aralığında) hesaplanmaktadır. Böylelikle aynı warp'ta yer alan thread'lerin global thread numarası ardışık şekilde sıralanmaktadır. Bir warp'taki thread sayısı 32 olduğu için ve segment büyüklüğü 128 bayt olduğu için ardışık sıralanan bu global thread numaraları ile thread'ler aynı segmente erişmektedir.
- Toplama işlemini döngü içinde 100 kez yapılarak bellek işlem sürelerinin kernel'in toplam çalışma süresine etkisi arttırılmakta böylelikle sonuçların daha tutarlı elde edilmesi sağlanmaktadır.
- non_coalesced_access kernel'inde A ve B değişkenlerinin işaret ettiği bellek bölgelerine tamamen dağınık (non-coalesced) bir şekilde erişim sağlanmaktadır. Global thread numaralarını kullanmak yerine döngünün her adımında her bir thread 0 ile toplam veri sayısı-1 aralığında rastgele bir sayı çekerek ilgili bellek bölgesine erişmeye çalışmaktadır. En kötü durumda aynı warp'teki her bir thread farklı segmentlere ait verileri okumaya çalışırsa warp'ın okuma işlemi 32 transaction'da tamamlanır.
- non_coalesced_access kernel'inde kullanılan curand metodu 0 ile $2^{32} - 1$ arasında rastgele sayılar üretmektedir. Mod işlemi alarak 0 ile toplam veri sayısı-1 aralığında sayı elde etmek pahalı bir işlemdir. Onun yerine 2^{32} sayısını toplam veri sayısı kadar eşit parçalara bölüyoruz. Her bir parçanın uzunluğu kernel'deki NP değişkeninde tutulmaktadır.

İlk olarak veri sayısını 32768 alarak her iki durumun çalışma zamanını ve transaction sayılarını ölçelim. Transaction sayılarını hesaplamak için profiler'daki aşağıdaki iki metriği kullanalım:

- **gld_transactions:** Global bellekten yapılan okuma (load) işlemlerinin (transaction) toplam sayısını gösterir.
- **gld_transactions_per_request:** Her bir global bellek okuma isteği başına gerçekleştirilen global bellek okuma işlemlerinin (transaction) ortalama sayısını ifade eder.

```
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
```

Bu örnekte toplam warp sayımız 1024'tür. Toplam transaction sayılarına (A + B için) bakarsak coalesced erişim durumunda her bir warp 1 transaction'da bellekten okuma işlemi yapmakta ve döngü boyunca toplamda 100 kez aynı işlemi tekrarlamaktadır. 1024 warp A için 102400 ve B için 102400 olmak üzere toplamda 204800 bellekten okuma işlemi yapmıştır. Bir warp her zaman bellekten okuma işlemini 1 transaction'da yaptığı için ortalama transaction sayısı da 1 çıkmaktadır.

Toplam transaction sayılarına (A + B için) bakarsak non-coalesced erişim durumunda her bir warp 32 transaction'a kadar bellekten okuma işlemi yapmakta ve döngü boyunca aynı işlemi toplamda 100 kez tekrarlamaktadır. A ve B için toplamda 6461956 bellekten okuma işlemi yapılmıştır. Bir warp bellekten okuma işlemini 32 transaction'a kadar yaptığı için ortalama transaction sayısı 30.633514 olarak çıkmıştır. Bu değerin 32'e yakın çıkması dağınık erişimi net bir şekilde göstermektedir.

Çalışma sürelerini inceleyince coalesced erişim non-coalesced erişme göre 14.9143x daha hızlı çalışmaktadır. Bu fark oldukça önemli bir süre farkıdır. Aynı deneyi bu sefer veri sayısını 4194304 alarak gerçekleştirelim:

```
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
```

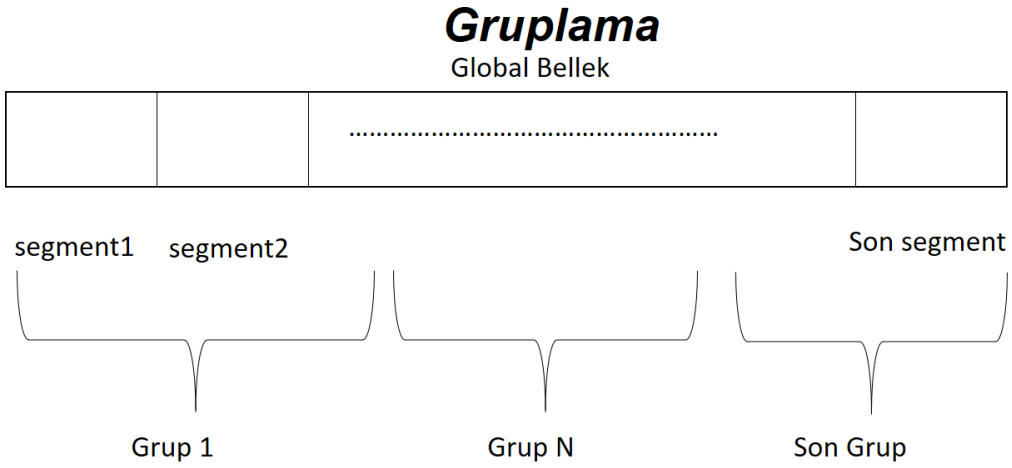
Bu örnekte toplam warp sayımız 131072'dir. Toplam transaction sayılarına (A + B için) bakarsak coalesced erişim durumunda her bir warp yine 1 transaction'da bellekten okuma işlemi yapmakta ve döngü boyunca toplamda 100 kez aynı işlemi tekrarlamaktadır. 131072 warp A için 13107200 ve B için 13107200 olmak üzere toplamda 26214400 bellekten okuma işlemi yapmıştır. Bir warp bellekten okuma işlemini her zaman 1 transaction'da yaptığı için ortalama transaction sayısı da yine 1 çıkmaktadır.

Toplam transaction sayılarına (A + B için) bakarsak non-coalesced erişim durumunda her bir warp 32 transaction'a kadar bellekten okuma işlemi yapmakta ve döngü boyunca toplamda 100 kez aynı işlemi tekrarlamaktadır. A ve B için toplamda 839546946 bellekten okuma işlemi yapılmıştır. Bir warp

bellekten okuma işlemini 32 transaction'a kadar yaptığı için ortalama transaction sayısı 31.093373 olarak çıkmıştır. Bu değerin 32'e yakın çıkması dağınık erişimi net bir şekilde göstermektedir.

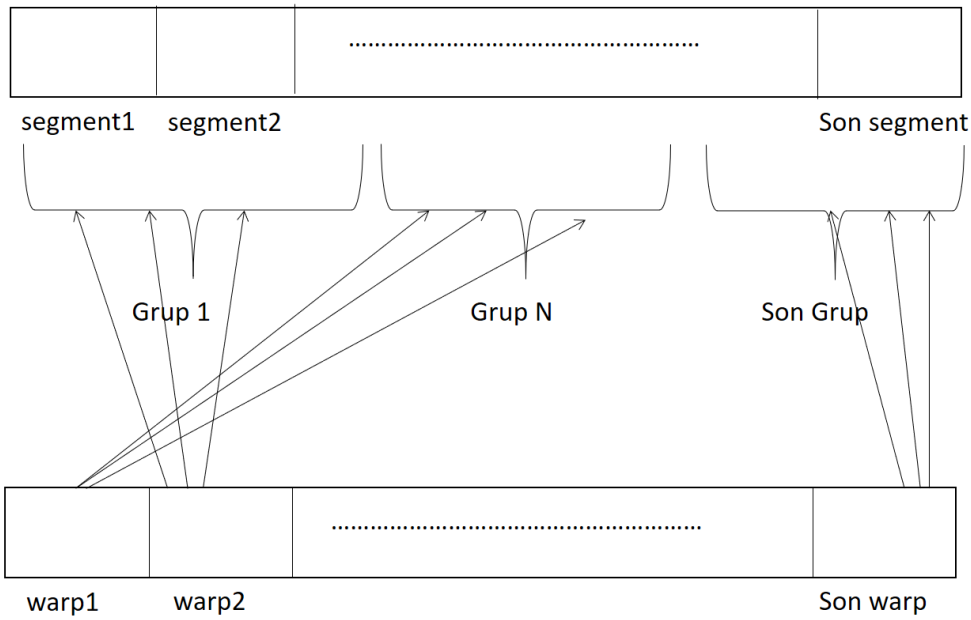
Çalışma sürelerini inceleyince coalesced erişim non-coalesced erişme göre 52.2214x daha hızlı çalışmaktadır. Bu fark oldukça önemli bir süre farkıdır.

Gerçek hayattaki problemlerde aynı warp'taki tüm thread'leri aynı segment üzerinde işlem yapmaya zorlamak bazen beklenilen sonuçları elde etmeye yetmeyebilir. Bu iki durumu uç durumlar olarak kabul edip ikisinin arasında olan bir erişim düzeni oluşturabiliriz. Bu düzende problemin çalışma süresi (hız) ile problemin çıktı kalitesi arasında bir ödünleşme sunulabilir. Bunu sağlamak için global bellekteki segmentleri gruplandırabiliriz (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)). Gruplar birbirini takip eden seğmenlerden oluşacaktır. Aşağıdaki şekilde bu yaklaşım gösterilmektedir:



- Gruplar birbirini takip eden segmentlerden oluşmaktadır.
- Bir gruptaki segment sayısı 1 ile veri sayısı/32 arasında olmaktadır (32 bir segmentteki veri sayısıdır).

Bu yaklaşımda bir warp'taki tüm thread'ler aynı gruptaki segmentlere erişmektedirler. Eğer grupların segment sayısını 1 olarak alırsak segment sayısı kadar grup olur ve coalesced erişim durumu gerçekleşmiş olur. Bu durumda tüm thread'ler aynı segmente erişir. Eğer grupların segment sayısı tüm segmentlerin sayısına eşit olursa 1 tane grup olur ve tüm segmentleri içerir. Bu durumda non-coalesced erişim durumu gerçekleşmiş olur. Grupların segment sayısını bu iki durumun arasında bir değer olarak belirlersek programın çalışma süresi ile problemin çıktı kalitesi arasında bir ödünleşme gerçekleşmiş olur. Bu yaklaşımda her warp için önce rastgele bir grup belirlenmekte sonrasında warp'taki thread'ler rastgele belirlenen bu grup içinde yer alan segmentlerdeki verilere rastgele erişmektedir. Örnek bir görsel aşağıda gösterilmektedir:



Bu yaklaşımın kodunun bir kısmı aşağıda verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Memory Coalesced Access isimli kodu indirebilirsiniz. Bu kodda kullanılmış olan random number generator'ların kodlarını içeren "RNG_functions.cuh" isimli başlık dosyasına Yardımcı Dosyalar bölümünde verilen RNG Functions Header isimli dosyayı indirerek erişebilirsiniz.

```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_grou
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
        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
}
```

- Gruptaki segment sayısı 16 olarak belirlenmiştir
- Dolayısıyla bir warp global bellekten okuma işlemini en fazla 16 transaction'da yapacaktır

Kod ile önemli kısımların açıklamalarını maddeler halinde yapalım:

- segment_size değişkeni bir segmentin bayt cinsinden büyüklüğünü tutmaktadır. Tesla K40 kartında bu değerin 128 olduğunu öğrenmiştik.
- group_size değişkeni bir gruptaki toplam veri sayısını tutmaktadır. Bu değer 16*(segment_size/4) formülüyle hesaplanmaktadır. Tam sayı türünden verilerle işlem yaptığımız için segment_size/4 bir segmentteki veri sayısını vermektedir. 16 ile çarpınca ardışık 16 tane segmentteki toplam veri sayısı hesaplanmış olmaktadır. Dolayısıyla bu örnekte bir gruptaki toplam veri sayısı 512 olmaktadır.
- group_count değişkeni girilen veri büyüklüğüne göre oluşan grup sayısı bilgisini tutmaktadır.
- semi_coalesced_access kernel'inde kullanılan curand metotları 0 ile $2^{32} - 1$ arasında rastgele sayılar üretmektedir. Mod işlemi alarak 0 ile toplam grup sayısı-1 aralığında sayı elde etmek pahalı bir işlemdir. Onun yerine 2^{32} sayısını toplam grup sayısı kadar eşit parçalara bölüyoruz. Her bir parçanın uzunluğu kernel'deki NPP_group_count değişkeninde tutulmaktadır. Sonrasında her warp için rastgele bir grup numarası belirlenip GN değişkeninde saklanmaktadır. İlgili warp'taki tüm thread'ler rastgele seçilmiş bu grup üzerindeki verileri okumaktadır.
- Thread'ler önce 0 ile grup eleman sayısı – 1 arasında rastgele bir indeks belirleyip sonrasında bu indeks üzerinden kendi grubundaki ilgili indekse erişmek için kaydırma işlemi yapmaktadır. Mod işlemi pahalı bir işlem olduğu için 2^{32} sayısını toplam grup eleman sayısı kadar eşit parçalara bölüyoruz. Her bir parçanın uzunluğu kernel'deki NPP_group_size değişkeninde tutulmaktadır. Not: Kaydırma işlemindeki GS değişkeni group_size değerine sahiptir.
- Çalışan kodda yer alan generator ile alakalı RNGs isimli nesne non-coalesced erişimde thread'lerin global bellekte yer alan elemanlara rastgele erişebilmeleri için kullanılmaktadır. RNGs1 isimli nesne ise semi-coalesced erişimde thread'lerin kendi warp'ı için rastgele seçilmiş gruptaki elemanlara rastgele erişebilmeleri için kullanılmaktadır. RNGs2 isimli nesne ise semi-coalesced erişimde her warp için rastgele bir grup seçilmesi için kullanılmaktadır. Bu rastgele seçimin warp bazında olabilmesi için curand_init fonksiyonu ile generator'ı oluştururken o warp içindeki tüm thread'lerin aynı sequence değerine sahip olmaları sağlanmıştır. Böylelikle ilgili warp'taki tüm thread'ler aynı gruptaki elemanlara rastgele erişmektedirler.

Önceki deneyi gruplama tekniğini de ekleyerek tekrar yapalım. Gruplardaki segment sayısı 16 olmaktadır. İlk olarak veri sayısını 32768 alarak her üç durumun çalışma zamanını ve transaction sayılarına ölçelim. Transaction sayılarını hesaplamak için profiler'da yine aynı iki metriği kullanalım:

```
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
```

Sonuçlar coalesced erişim ve non-coalesced erişim için önceki deneyle çok benzer çıkmıştır. semi-coalesced erişim durumu için toplam transaction sayılarına (A + B için) bakarsak her bir warp 16

transaction'a kadar bellekten okuma işlemi yapmakta olup döngü boyunca toplamda 100 kez aynı işlemi tekrarlamaktadır. A ve B için toplamda 2870920 bellekten okuma işlemi yapmaktadır. Bir warp bellekten okuma işlemini 16 transaction'a kadar yaptığı için ortalama transaction sayısı 13.414511 olarak çıkmıştır. Bu değerin 16'e yakın çıkması kısmi dağınık erişimi net bir şekilde göstermektedir.

Çalışma sürelerini inceleyince semi-coalesced erişim non-coalesced erişme göre 6.96495x daha hızlı çalışmaktadır. Böylelikle problemin sonuçlarının kalitesini arttırma şansımız yükselirken süreden de önemli ölçüde kazanç sağlıyoruz. Aynı deneyi bu sefer veri sayısını 4194304 alarak gerçekleştirelim:

```
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
```

Sonuçlar coalesced erişim ve non-coalesced erişim için önceki deneyle çok benzer çıkmıştır. semi-coalesced erişim durumu için toplam transaction sayılarına (A + B için) bakarsak her bir warp 16 transaction'a kadar bellekten okuma işlemi yapmakta olup döngü boyunca toplamda 100 kez aynı işlemi tekrarlamaktadır. A ve B için toplamda 367412642 bellekten okuma işlemi yapmaktadır. Bir warp bellekten okuma işlemini 16 transaction'a kadar yaptığı için ortalama transaction sayısı 13.412134 olarak çıkmıştır. Bu değerin 16'e yakın çıkması kısmi dağınık erişimi net bir şekilde göstermektedir.

Çalışma sürelerini inceleyince semi-coalesced erişim non-coalesced erişme göre 22.2288x daha hızlı çalışmaktadır. Böylelikle problemin sonuçlarının kalitesini arttırma şansımız yükselirken süreden de önemli ölçüde kazanç sağlıyoruz.

2. Warp Divergence (Ayrışması):

'If-else' gibi bazı yapılar bir warp için tek bir komut (instruction) olarak sayılmaktadır. Hatırlarsak bir warp'ın bir komutu tamamlaması için o warp'taki tüm thread'lerin aynı komutu tamamlaması gerekirdi. Eğer warp'taki thread'lerin bazıları 'if' koşulunu bazıları 'else' koşulunu çalıştırırsa bu işlemler serileşir. Örneğin:

if(tid %2 == 0)//tid global thread numarasıdır

else

Bu durumda warp'taki global thread numarası çift olanlar 'if' koşulunu çalıştırır. Bu aşamada global thread numarası tek olanlar bekleme durumunda olur. Daha sonra global thread numarası tek olanlar 'else' koşulunu çalıştırır ve çift olanlar bekleme durumunda olur. Dolayısıyla bir warp'taki tüm thread'lerin aynı koşulu çalıştırması serileşmeyi önleyecektir. Bu durumu sağlamanın yollarından bir tanesi 'if' koşulunda global thread numarası yerine global thread numarasını warp büyüklüğüne bölüp elde edilen değeri kullanabiliriz:

if((tid/32) %2 == 0)//tid global thread numarasıdır

else

Bu değer aslında bir anlamda warp numarası gibi düşünülebilir. Böylelikle tüm thread'ler aynı değere sahip olur. Warp numarası çift olanlar 'if' koşulunu çalıştırır, tek olanlar 'else' koşulunu çalıştırır. Bu iki durumu çeşitli vektör işlemlerinin yapıldığı aşağıda kodunun bir kısmı verilen bir örnek üzerinde inceleyelim. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Warp Divergence isimli kodu indirebilirsiniz.

```
__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
    }
}
```

- Koşullar warp numarasına göre dağıtılmaktadır
- Birinci warp toplama, ikinci warp çıkarma, üçüncü warp çarpma, dördüncü warp bölme yapmaktadır

```
__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
    }
}
```

- Koşullar thread numarasına göre dağıtılmaktadır
- Birinci thread toplama, ikinci thread çıkarma, üçüncü thread çarpma, dördüncü thread bölme yapmaktadır

```
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
}
```

- Yeterince değişik yol var (4)
- Komutların yeterince tekrarı var (100)
- Bellek işlemleri coalesced! Dolayısıyla warp divergence toplam çalışma süresini domine ediyor!
- Kernel çalışma zamanlarını ölçmek için **cudaEventRecord** kullanıldı

Kodla ilgili bazı açıklamaları maddeler halinde yapalım:

- warp_no_divergence kernel'inde koşul olarak tid/32 kullanılmaktadır. Bu durumda thread'ler kendisine ait warp'ın numarasına göre ilgili koşulu çalıştıracaktır. Warp numaralarının 4'e göre modu alınmıştır. Mod değeri 0 olan warp'ların thread'leri toplama, 1 olan warp'ların thread'leri çıkarma, 2 olan warp'ların thread'leri çarpma, 3 olan warp'ların thread'leri bölme işlemi yapmaktadır. Böylelikle bir warp'taki thread'ler hep aynı koşulu çalıştırmaktadır.
- warp_divergence kernel'inde ise koşul olarak tid kullanılmaktadır. Bu durumda thread'ler kendi global thread numaralarına göre ilgili koşulu çalıştıracaktır. Global thread numaralarının 4'e göre modu alınmıştır. Mod değeri 0 olan thread'ler toplama, 1 olan thread'ler çıkarma, 2 olan thread'ler çarpma, 3 olan thread'ler bölme işlemi yapmaktadır. Bu durumda bir warp içindeki thread'ler dört farklı koşulu da çalıştırmakta ve işlemler serileşmektedir.

Bu iki senaryoyu hem toplam çalışma süresi açısından hem de aşağıda verilen metrik açısından kıyaslayacağız:

- **warp_execution_efficiency:** Bir SM'de desteklenen warp başına maksimum thread sayısına kıyasla, warp başına ortalama aktif thread sayısının oranının yüzde (%) cinsinden ifadesi.

```
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%
```

warp_no_divergence kernel'inde hiç warp ayrışması olmadığı için warp execution efficiency %100 çıkmıştır. warp-divergence kernel'inde olan warp ayrışmasından dolayı warp execution efficiency %34.8'lere kadar düşmüştür. Toplam çalışma sürelerine de bakınca bu orana benzer şekilde 3.10078x hızlanma gerçekleşmiştir. Bu sonuçlar warp divergence probleminin performansa olan olumsuz etkisini ortaya açıkça koymaktadır.

3. Occupancy ve SM Efficiency:

Occupancy, bir SM üzerinde aynı anda aktif olarak çalışan warp sayısının, o SM'in desteklediği maksimum warp sayısına oranıdır. SM'lerin kaynak limitleriyle bağlantılıdır. Bu limitlerin bazıları şunlardır:

- Bir SM'de yer alan maksimum thread sayısı (2048)
- Bir blokta yer alan maksimum thread sayısı (1024)
- Bir SM'de yer alan maksimum blok sayısı (16)
- Shared memory (ortak bellek) ve register (yazmaç) kısıtlamaları
    - **--ptxas-options=-v** derleyici bayrağı ile shared memory ve register kullanımı ile ilgili bilgiler elde edilebilir.

Ana amaç en yüksek occupancy değerlerini elde etmek için bir blokta yer alacak thread sayısının değerini en uygun olarak seçmektir. Eğer bir bloktaki thread sayısını 64 olarak seçersek bir SM'de 16 × 64 (1024) thread aktif olarak çalışabilmektedir. Bu durumda toplam çalışabilecek thread sayısının yarısı kadar thread aktif olarak çalışmış olmaktadır. Dolayısıyla teorik olarak %50 occupancy elde edilir. Eğer bir bloktaki thread sayısını 128 olarak seçersek bir SM'de 16 × 128 (2048) thread aktif olarak çalışabilmektedir. Bu durumda aynı anda çalışabilecek tüm thread'ler aktif olarak çalışmış olacaktır. Teorik occupancy değeri de %100'e ulaşmış olacaktır. Bir bloktaki thread sayısını 1024 seçer isek yine %100 teorik occupancy değerini elde edebiliriz. Fakat bu durumda bir SM'de aktif olarak çalışan blok sayısı 2'ye düşmüş olacaktır. Deneylerde bir önceki bölümde gördüğümüz warp divergence durumu olmayan çeşitli vektör işlemlerinin yapıldığı kernel'i kullanacağız. Occupancy ve SM efficiency değerlerini ölçmek için bir blokta yer alan thread sayıları ile ilgili iki farklı senaryo oluşturalım:

Hatırlatma: Bloklar SMX'lere çalıştırılmak üzere gönderilmektedir. Tesla K40'ta 15 tane SMX vardır. Biz deneylerde 16 SMX var gibi düşüneceğiz.

1. İlk senaryoda bir bloktaki thread sayısını veri sayısına eşitliyoruz. Bu durumda kernel'de sadece 1 blok oluşmaktadır. Bir bloktaki thread sayısı en fazla 1024 olduğu için 2048 veri sayısından itibaren tüm veri sayılarında thread sayısı 1024 olarak alınmaktadır. Dolayısıyla oluşan blok sayısı da aynı oranda artmaktadır.

2. İkinci senaryoda ise bir bloktaki thread sayısını belirlerken thread'leri mümkün olan en fazla sayıda bloka paylaştırmaya çalışmaktayız. Böylelikle thread'lerin farklı farklı SMX'lerde çalışmasını en baştan sağlamaya çalışıyoruz. Bu doğrultuda bir bloktaki thread sayısını veri sayısı 1024 olana kadar 32 olarak alıyoruz. 1024'ten itibaren thread sayısını veri sayısının artış oranı ile aynı olacak şekilde arttırmaktayız. Bir bloktaki thread sayısı en fazla 1024 olduğu için 32768 veri sayısından itibaren tüm veri sayılarında thread sayısı 1024 olarak alınmaktadır. Dolayısıyla oluşan blok sayısı da aynı oranda artmaktadır.

Deneylerde kullanılan uygulamanın kodunun bir kısmı aşağıda verilmektedir. Kodun tamamına ulaşmak için Örnek Kodlar bölümünde verilen Occupancy isimli kodu indirebilirsiniz.

```
__global__ void vector_add(float *A,float *B,float *C)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<1000000;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
    }
}

int main(int argc, char **argv)
{
    int data_size;//Data size

    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    int NTB;//Number of threads per block
    if(data_size <= 1024)//Scenario 1 - Set NTB to data_size until 2^
        NTB = data_size;
    else
        NTB = 1024;
    dim3 threadsPerBlockS1(NTB);//Number of threads in a block
    dim3 numBlocksS1(data_size/NTB);//Number of blocks in a grid

    vector_add<<<numBlocksS1,threadsPerBlockS1>>>(A_GPU,B_GPU,C_GPU);

    if(data_size <= 512)//Scenario 2 - Increase NTB to its double
        NTB = 32;
    else if(data_size == 1024)
        NTB = 64;
    else if(data_size == 2048)
        NTB = 128;
    else if(data_size == 4096)
        NTB = 256;
    else if(data_size == 8192)
        NTB = 512;
    else
        NTB = 1024;
    dim3 threadsPerBlockS2(NTB);//Number of threads in a block
    dim3 numBlocksS2(data_size/NTB);//Number of blocks in a grid

    vector_add<<<numBlocksS2,threadsPerBlockS2>>>(A_GPU,B_GPU,C_GPU);

}
```

- Iterasyon sayısı 1000000 olarak belirlendi. Böylelikle 'if-elseif' yapısı çalışma zamanını domine etmektedir.
- warp divergence durumu yok.
- Kernel çalışma zamanını ölçmek için **cudaEventRecord** komutu kullanılmaktadır.

Senaryo1 kodu

Senaryo2 kodu

Bu iki senaryo ile ilgili bilgiler aşağıdaki tabloda verilmektedir:

Veri sayısı	Senaryo 1			Senaryo 2		
	Thread sayısı	Blok sayısı	T. Occup.	Thread sayısı	Blok sayısı	T. Occup.
32	32	1	%25	32	1	%25
64	64	1	%50	32	2	%25
128	128	1	%100	32	4	%25
256	256	1	%100	32	8	%25
512	512	1	%100	32	16	%25
1024	1024	1	%100	64	16	%50
2048	1024	2	%100	128	16	%100
4096	1024	4	%100	256	16	%100
8192	1024	8	%100	512	16	%100
16384	1024	16	%100	1024	16	%100
32768	1024	32	%100	1024	32	%100
65536	1024	64	%100	1024	64	%100

Bir blokta 32 thread yer alıyorsa teorik occupancy değerinin %25, 64 thread yer alıyorsa %50, 128 ve üzeri thread yer alıyorsa %100 olduğunu öğrenmiştik. Bu durumda senaryo 1 ile daha iyi teorik occupancy değerleri elde etmekteyiz. Fakat teorik occupancy değerinin artması her zaman daha iyi çalışma süreleri elde edeceğimiz anlamına gelmemektedir. SMX'lerin etkin biçimde kullanılması da uygulamanın çalışma süresini iyileştiren temel performans faktörlerinden biridir. Bloklar SMX'lere çalışmak üzere gönderilmekteydi. SMX'lere eşit sayıda blok göndermek tüm SMX'lerin eş zamanlı olarak çalışmalarına olanak sağlayacaktır. İkinci senaryoda bu durum amaçlanmaktadır. Bir bloktaki thread sayısı her SMX'e bir tane blok gönderilecek şekilde ayarlanmaya çalışılmaktadır. Yeterince blok olmadığında mümkün olan en yüksek sayıda SMX aktif olarak çalışmaktadır. Bu senaryoda teorik occupancy azalırken SMX verimliliği artmaktadır. Bu durum kernel'in çalışma süresine olumlu katkıda bulunacak ve bize teorik occupancy değerinin artmasının her zaman kernel çalışma süresinin iyileşmesinin anlamına gelmediğini gösterecektir. Deneylerde bu ölçümleri yapmak için aşağıdaki iki metriği kullanacağız:

1. **sm_efficiency:** GPU üzerindeki tüm çoklu işlemciler (Streaming Multiprocessors – SMX) için ortalama alınarak, bir SMX üzerinde en az bir warp'ın aktif olduğu sürenin yüzdesini gösterir.
    a. Her bir çoklu işlemcinin (SMX) aktif çalışma süresinin GPU'nun toplam çalışma süresine oranının hesaplanması ve bu oranların ortalamasının alınmasıyla elde edilir.
2. **achieved_occupancy:** Bir SMX üzerinde, aktif döngüler (cycle) başına ortalama aktif warp sayısının, o SMX'in desteklediği maksimum warp sayısına oranını ifade eder.
    a. achieved_occupancy teorik occupancy değerini hiçbir zaman geçemez.

Deneysel sonuçlar aşağıdaki tabloda verilmektedir. Bu tabloda değişik veri büyüklükleri için her bir senaryoda kullanılan blok sayısı, bir bloktaki thread sayısı ve elde edilen teorik occupancy değerleri ile metriklerden elde edilen SMX efficiency, achieved_occupancy değerleri ve kernel çalışma zamanı verilmektedir.

Veri Sayısı	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
	Thread sayısı		Blok sayısı		SM Efficiency		Teorik Occup.		Achieved Occup.		Çalışma zamanı	
32	32	32	1	1	6.54%	6.55%	25%	25%	1.5%	1.5%	0.35	0.35
64	64	32	1	2	6.54%	12.46%	50%	25%	2.9%	1.5%	0.41	0.41
128	128	32	1	4	6.54%	18.83%	100%	25%	4.3%	1.5%	0.69	0.69
256	256	32	1	8	6.54%	37.17%	100%	25%	8.8%	1.5%	0.70	0.69
512	512	32	1	16	6.54%	68.48%	100%	25%	17%	1.6%	0.71	0.69
1024	1024	64	1	16	6.54%	78.32%	100%	50%	35%	2.8%	0.74	0.70
2048	1024	128	2	16	13.15%	96.32%	100%	100%	35%	4.7%	0.75	0.70
4096	1024	256	4	16	26.26%	95.91%	100%	100%	35%	9.4%	0.75	0.72
8192	1024	512	8	16	52.41%	95.10%	100%	100%	35%	18%	0.75	0.74
16384	1024	1024	16	16	83.29%	83.28%	100%	100%	39%	39%	0.91	0.91
32768	1024	1024	32	32	62.07%	62.11%	100%	100%	72%	72%	1.6	1.6
65536	1024	1024	64	64	72.80%	72.80%	100%	100%	77%	75%	2.49	2.49

Yeşil renkli değerler ilgili çıktı için o senaryonun diğer senaryoya göre daha iyi olduğunu göstermektedir. Sarı renkli değerler ise her iki senaryoda da parametre değerlerinin aynı olduğunu ve bu nedenle çıktıların neredeyse aynı olduğunu ifade etmektedir. SMX efficiency sonuçlarını inceleyince S2'nin thread sayısının S1'in thread sayısına eşit olmadığı tüm durumlarda S2 daha iyi sonuç elde etmiştir. Teorik occupancy sonuçlarını inceleyince S1 birçok durumda S2'den daha iyi sonuç elde etmiştir. Ayrıca achieved occupancy sonuçlarında S1'in thread sayısının S2'in thread sayısına eşit olmadığı tüm durumlarda S1 daha iyi sonuç elde etmiştir. Kernel çalışma zamanlarına bakınca S1 daha iyi occupancy değerlerine sahip olduğu halde S2 birçok durumda S1'den daha iyi sürelere ulaşmıştır. Dolayısıyla occupancy kavramını incelerken SMX'lerin verimliliğini mutlaka göz önünde bulundurmamız gerekmektedir.

Diğer taraftan sonuçlarda achieved_occupancy değerlerinin teorik occupancy değerlerinin oldukça altında olduğu görülmektedir. Genel olarak bu durumun sebepleri şunlar olabilir (Ref: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm):

1. **Blok içi dengesiz iş yükü:**
    a. Bir blok içindeki warp'ların iş yükleri dengesizdir.
2. **Bloklar arası dengesiz iş yükü:**
    a. Grid içindeki blokların iş yükleri dengesizdir.
3. **Yetersiz sayıda blok başlatılması:**
    a. Bir SMX üzerinde aynı anda aktif olabilecek maksimum blok sayısından daha az blok çalıştırılması.
4. **Son dalganın kısmi olması (Partial last wave):**
    a. Bir SMX üzerinde aynı anda aktif olabilecek maksimum warp sayısına ulaşılamaması.