DEBUGGING VE PROFILING

1. CUDA Debugging:

CUDA debugger, CUDA C/C++ kodunuzdaki hataları bulup düzeltmenizi sağlayan bir hata ayıklama (debugging) aracıdır. CUDA'da yazdığınız kodun bir kısmı CPU tarafından çalıştırılırken, veri paralelliği içeren kısımlar GPU üzerinde binlerce paralel iş parçacığıyla (thread) aynı anda yürütülür. Bu yüzden geleneksel CPU odaklı debugger'lar GPU'nun paralel mimarisini anlayamaz. Bu yüzden GPU'nun paralel yapısını destekleyen özel bir debugger gerekir. CUDA debugger ile aşağıdaki işlemleri yapabilirsiniz:

- ✓ **Breakpoint (durdurma noktası) koyma:** Belirli satırda kodu durdurma.
- ✓ **Step-by-step yürütme:** Kodu satır satır çalıştırıp davranışı inceleme.
- ✓ **Değişken ve bellek değerlerini gözlemleme:** CPU ve GPU'daki değerlerini değerleri görme.
- ✓ **GPU kernel'ların çalışmasını takip etme:** Kernel kodunun ne zaman başlayıp bittiğini, nasıl davrandığını görme.

Bazı temel CUDA debugging araçları şunlardır:

- ▶ CUDA-MEMCHECK
- ▶ CUDA-GDB
- ▶ Nsight

1.1. CUDA-MEMCHECK:

GPU bellek kullanımındaki yanlışları bulup bildirerek CUDA kodunuzun doğruluğunu sağlamaya yardımcı olan bir araçtır. Bu araç çalıştırdığınız CUDA programını izleyerek (monitoring) çeşitli denetimler yapmaktadır. CUDA-MEMCHECK aracı CUDA toolkit ile birlikte yüklenmektedir. CUDA'yı destekleyen tüm platformlarda çalışır. SM sürümü 3.5 ve üzeri olan tüm CUDA uyumlu GPU'lar tarafından desteklenmektedir. Sanal GPU'larda desteklenmemektedir. CUDA MEMCHECK içinde yer alan araçlar aşağıda belirtilmektedir. Bu dokümanda bu araçlardan Memcheck ve Initcheck kullanımına ilişkin örnekler verilecektir:

- ➢ **Memcheck:** Bellek erişim hatalarını (memory access error) ve bellek sızıntılarını (memory leak) tespit eden araçtır.
- ➢ **Racecheck:** Paylaşılan bellek (shared memory) üzerindeki veri erişim çakışmalarını / tehlikelerini (data access hazard) tespit eden araçtır.
- ➢ **Initcheck:** Henüz hiç değer atanmamış (uninitialized) GPU ana belleğine (device global memory) yapılan erişimleri tespit eden araçtır.
- ➢ **Synccheck:** İş parçacığı (thread) senkronizasyonu (synchronization) ile ilgili tehlikeleri (hazard) tespit eden araçtır.

1.1.1 Memcheck: Memcheck aracı, CUDA uygulamaları çalışırken ortaya çıkan hataları tespit eden bir araçtır. Memcheck aracının denetleyip raporlayabildiği hata türleri aşağıda özet olarak verilmektedir:

- **Bellek erişim (memory access) hataları:** Sınır dışı (out of bounds) yada yanlış hizalanmış (misaligned) bellek erişimi.
- **Malloc/Free hataları:** CUDA kernel içinde bellekte yer ayırma yada bellek silme işlemleri yapılırken malloc/free komutlarının hatalı kullanılması.

- **cudaMalloc bellek sızıntıları (memory leaks):** cudaMalloc ile GPU belleğinde ayrılmış alanın cudaFree komutu ile temizlenmemesi.
- **Device Heap bellek sızıntıları (memory leaks):** GPU kodunda malloc ile ayrılan bellek alanının free komutu ile temizlenmemesi.
- **CUDA API hataları:** API çağrı hatalarını raporlar.
- **Donanım istisnaları (hardware exception):** Donanımın hata raporlama mekanizması tarafından bildirilen hatalardır.

Örnek1: Bellek erişim hatasının (memory access error) ortaya çıktığı bu örnekte dizinin sınırları dışında bir bölgeye erişilmeye çalışılmaktadır. CUDA kodu aşağıda verilmektedir:

```
__global__ void memory_access_error(int *A)
{
    *(A + 3) = 100;  ──────────────────────►  sınır dışı hatası
}

void run_access_error(int *A_GPU)
{
    printf("Running memory_access_error kernel\n");
    memory_access_error<<<1,1>>>(A_GPU);
    printf("Ran memory_access_error kernel\n");
    printf("cudaGetLastError()-1 = %s\n",cudaGetErrorString(cudaGetLastError()));
    printf("Error: %s\n", cudaGetErrorString(cudaDeviceSynchronize()));
    printf("cudaGetLastError()-2 = %s\n",cudaGetErrorString(cudaGetLastError()));
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    run_access_error(A_GPU);
    cudaFree(A_GPU);
}
```

Bu kodu aşağıdaki gibi derleyip çalıştırabiliriz:

$ nvcc -o memcheck1 memcheck1.cu

$ ./memcheck1

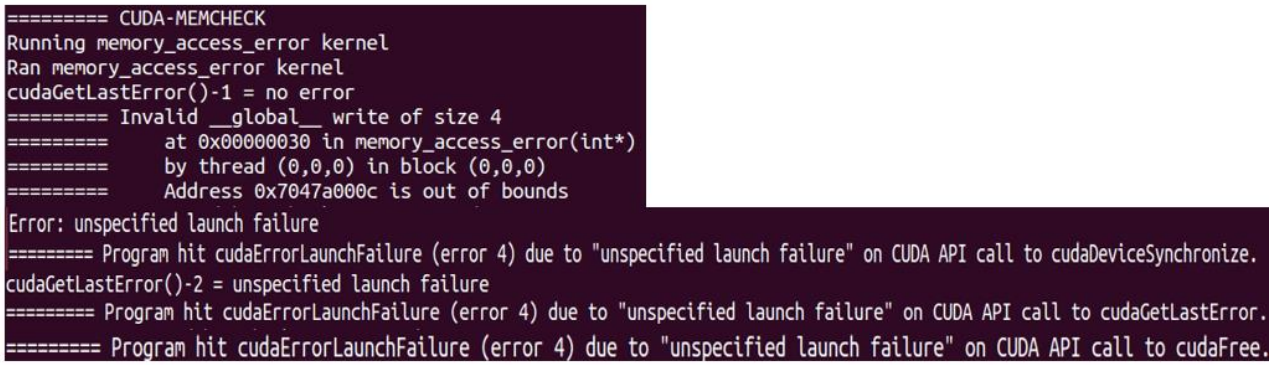
Ekran çıktısı ise aşağıdaki gibi olacaktır:

Running memory_access_error kernel
Ran memory_access_error kernel
cudaGetLastError()-1 = no error
Error: no error
cudaGetLastError()-2 = no error

Koddaki hatayı ve hatanın açıklamasını döndüren fonksiyonlar her hangi bir hata yok şeklinde değerler döndürdüler. Aslında hata oluşmuştu ama yakalayamadılar. Dolayısıyla kodumuzda gözümüzden kaçan bir bellek erişim hatası var ise bu hatayı cuda-memcheck aracını kullanarak tespit etmeliyiz. Kodumuzu terminalde cuda-memcheck aracıyla aşağıdaki gibi çalıştırabiliriz:

$ cuda-memcheck memcheck1

Ekran çıktısı aşadağıdaki gibi olacaktır. GPU'nuzun SM sürümüne göre ekran çıktısında değişiklikler olabilir:



cuda-memcheck aracı tarafından ekranda gösterilen bilgileri tek tek inceleyelim:

- **cudaGetLastError()-1 = no error:** Kernel çalışmaları asenkron olduğu için kernel'de hata oluşmadan önce ilk cudaGetLastError() çalışmaktadır ve çıktı no error olmaktadır.
- **Invalid __global__ write of size 4:** GPU global belleğine geçersiz bir yazma işlemi yapılmış demektir. 4 baytlık bir veri yazılmaya çalışılmış. Örneğimizdeki veri 4 bayt yer kaplayan **int** türünde bir veridir.
- **at 0x00000030 in memory_access_error(int*):** Hataya sebep veren komutun kernel içindeki konumunu gösterir. Burada hata veren komut memory_access_error(int*) kernel'i içinde 0x00000030 talimat adresindeki (instruction offset) komuttur.
- **by thread (0,0,0) in block (0,0,0):** Hatanın hangi thread ve hangi block tarafından yapıldığını belirtmektedir. Yani ilk block'taki ilk thread bu hatalı bellek erişimini yapmış.
- **Address 0x7047a000c is out of bounds:** Yazılmaya çalışılan bellek adresi (0x7047a000c) ayrılan bellek sınırlarının dışındadır.
- **Error: unspecified launch failure:** cudaDeviceSynchronize() komutu ile CPU, GPU'daki işlemler bitene kadar beklemektedir. Kernel'de hata oluşup kernel sonlandıktan sonra hata bu satırda yakalanmakta ve ekranda unspecified launch failure hatası gösterilmektedir. Hemen altında cuda-memcheck cudaErrorLaunchFailure isimli hata kodunu içeren mesajı ekranda göstermektedir.
- **cudaGetLastError()-2 = unspecified launch failure:** Kernel'de oluşan cudaErrorLaunchFailure hatası ikinci cudaGetLastError() tarafından da yakalanmaktadır ve ekranda unspecified launch failure hatası gösterilmektedir. Hemen altında cuda-memcheck cudaErrorLaunchFailure isimli hata kodunu içeren mesajı ekranda göstermektedir.
- **cudaFree() çalışması:** Kernel'de oluşan cudaErrorLaunchFailure hatası cudaFree tarafından da yakalanmaktadır. cuda-memcheck cudaErrorLaunchFailure isimli hata kodunu içeren mesajı ekranda göstermektedir.

Örnek2: Bellek erişim hatasının (memory access error) ortaya çıktığı bu örnekte hatalı hizalanmış bellek erişimi gerçekleştirilmeye çalışılmaktadır. CUDA kodu aşağıda verilmektedir:

```
__global__ void memory_access_error(int *A)
{
    *(int*) ((char*)&A[0] + 1) = 100;
}

void run_access_error(int *A_GPU)
{
    printf("Running memory_access_error kernel\n");
    memory_access_error<<<1,1>>>(A_GPU);
    printf("Ran memory_access_error kernel\n");
    printf("cudaGetLastError()-1 = %s\n",cudaGetErrorString(cudaGetLastError()));
    printf("Error: %s\n", cudaGetErrorString(cudaDeviceSynchronize()));
    printf("cudaGetLastError()-2 = %s\n",cudaGetErrorString(cudaGetLastError()));
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    run_access_error(A_GPU);
    cudaFree(A_GPU);
}
```

hatalı hizalanmış erişim

4'ün katları olmalı

4 baytlık int türünde veri tutan bir dizinin elemanlarına doğru şekilde erişebilmek için dizinin başlangıç adresinden itibaren 4'ün katları olan bayt bölgelerine (ör. 0, 4, 8, 12,…) erişmemiz lazım. Fakat bu örnekte 1. bayt bölgesine (yani 4 bayt yer kaplayan ilk tam sayı değerin 2. baytı) erişmeye çalıştığımızdan dolayı hata ile karşılaşıyoruz. cuda-memcheck ile çalıştırdıktan sonra aldığımız uyarı mesajları Örnek1'deki uyarı mesajları ile aynı çıkmaktadır. Tek farklı olan mesaj aşağıda verilmektedir:

- **Address 0x7047a0001 is misaligned:** Erişilmeye çalışılan bellek adresinin (0x7047a0001) ilgili veri türü için gerekli hizalamaya uymadığını belirtmektedir.

Örnek3: Bu örnekte free komutu ile daha önce serbest bırakılmış GPU bellek bölgesinin tekrardan serbest bırakılmasıyla Malloc/Free hatası ortaya çıkmaktadır. CUDA kodu aşağıda verilmektedir:

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    free(array);
    free(array);
}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Her thread GPU'nun heap bellek bölgesinde yer ayırmaktadır.

Hata! Önceden serbest bırakılan bellek bölgesi serbest bırakılmaya çalışılıyor

Bu kodda her bir iş parçacığı (thread) GPU'nun heap bellek bölgesinde malloc komutu ile yer ayırmaktadır. Sonrasında bu bellek bölgesini free komutu ile serbest bırakmaktadır. Daha sonra aynı bellek bölgesini tekrar serbest bırakmaya kalkınca hata oluşmaktadır. cuda-memcheck aracı ile kodu çalıştırdıktan sonra ekranda gösterilen bilgileri tek tek inceleyelim:

```
CUDA-MEMCHECK
Malloc/Free error encountered : Double free
    at 0x000691c0
    by thread (1,0,0) in block (0,0,0)
    Address 0x70509f970
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x60)
Malloc/Free error encountered : Double free
    at 0x000691c0
    by thread (0,0,0) in block (0,0,0)
    Address 0x70509f920
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x60)
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API cal
l to cudaDeviceSynchronize.
```

- **Malloc/Free error encountered : Double free:** Aynı bellek adresi iki kez free edilmeye çalışılmış uyarı mesajıdır.
- **at 0x000691c0:** Hataya sebep veren komutun kernel içindeki konumunu talimat adresi (instruction offset) olarak göstermektedir.
- **by thread (1,0,0) in block (0,0,0):** Hatanın hangi thread ve hangi block tarafından yapıldığını belirtmektedir. Burada ilk block'taki ikinci thread bu hatalı bellek erişimini yapmış.
- **Address 0x70509f970:** İkinci thread tarafından ikinci kez serbest bırakılmaya çalışılan bellek adresini göstermektedir.
- **Device Frame: malloc_free_error(void):** Hatanın GPU'daki malloc_free_error(void) isimli kernel içinde oluştuğunu gösterir.
- **by thread (0,0,0) in block (0,0,0):** Hatanın hangi thread ve hangi block tarafından yapıldığını belirtmektedir. Burada ilk block'taki ilk thread bu hatalı bellek erişimini yapmış.
- **Address 0x70509f920:** İlk thread tarafından ikinci kez serbest bırakılmaya çalışılan bellek adresini göstermektedir.
- **cudaDeviceSynchronize() çalışması:** cudaDeviceSynchronize() komutu ile CPU, GPU'daki işlemler bitene kadar beklemektedir. Kernel'de hata oluşup kernel sonlandıktan sonra hata bu satırda yakalanmaktadır. cuda-memcheck cudaErrorLaunchFailure isimli hata kodunu içeren mesajı ekranda göstermektedir.

Örnek4: Bu örnekte free komutu ile malloc tarafından döndürülmemiş bir GPU bellek bölgesi serbest bırakılmaya çalışılınca Malloc/Free hatası ortaya çıkmaktadır. CUDA kodu aşağıda verilmektedir:

```
__global__ void malloc_free_error()
{
    int* array = (int*)malloc(10*sizeof(int));
    array = array + 1;
    free(array);

}

int main()
{
    malloc_free_error<<<1,2>>>();
    cudaDeviceSynchronize();
}
```

Her thread GPU'nun heap bellek bölgesinde yer ayırmaktadır.

Hata! malloc() tarafından döndürülmemiş bellek bölgesi serbest bırakılmaya çalışılıyor

Bu kodda her bir iş parçacığı (thread) GPU'nun heap bellek bölgesinde malloc komutu ile yer ayırmaktadır. Sonrasında malloc ile döndürülen bellek adresini tutan pointer'ın değerini değiştirmektedir. Daha sonra pointer'ın yeni işaret ettiği adresi serbest bırakmaya kalkınca hata oluşmaktadır. cuda-memcheck aracı ile kodu çalıştırdıktan sonra ekranda gösterilen bilgileri inceleyelim. Birçok mesaj önceki örneğin hata mesajları ile aynı olup farklı olan hata mesajı aşağıda gösterilmektedir:

```
CUDA-MEMCHECK
Malloc/Free error encountered : Invalid pointer to free
    at 0x000691c0
    by thread (1,0,0) in block (0,0,0)
    Address 0x70509f974
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x40)
Malloc/Free error encountered : Invalid pointer to free
    at 0x000691c0
    by thread (0,0,0) in block (0,0,0)
    Address 0x70509f924
    Device Frame:malloc_free_error(void) (malloc_free_error(void) : 0x40)
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API cal
l to cudaDeviceSynchronize.
```

- **Malloc/Free error encountered : Invalid pointer to free:** Malloc tarafından döndürülmemiş bir bellek bölgesi serbest bırakılmaya çalışılırken oluşan hata mesajıdır.

Örnek5: Bu örnekte GPU belleğinde yer açılan bir bölge free komutu ile serbest bırakılmadan program sonlanıyorsa bellek sızıntısı hatası (leak error) ortaya çıkmaktadır. CUDA kodu aşağıda verilmektedir:

```
__global__ void error_free_error(int *A)
{
    A[0] = 1;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*3);
    error_free_error<<<1,1>>>(A_GPU);
    cudaDeviceReset();
}
```

A_GPU serbest bırakılmıyor. Bellek sızıntısına (leak error) sebep olur.

CUDA context'i temizlemek için kullanılır. Sızıntılar dahil her şeyi temizler. Ama leak error için önerilmez!

Bu kodu aşağıdaki gibi derleyip çalıştırabiliriz:

$ nvcc -o memcheck1 memcheck1.cu

$ cuda-memcheck --leak-check full memcheck1

cuda-memcheck, cudaDeviceReset() çağrılmadan önce serbest bırakılmamış bellek bölgelerini tespit ederek bellek sızıntı hatalarını (leak error) raporlayabilir. cudaDeviceReset() komutu, sızıntılar dâhil olmak üzere CUDA uygulamasına ait tüm GPU kaynaklarını serbest bırakarak CUDA context'i tamamen yok eder; ancak bu yöntem bellek yönetimi için önerilmez. Program çalışırken GPU belleğinde ayrılan bellek bölgeleri, ilgili işlemler tamamlandıktan sonra cudaFree() ile serbest bırakılmazsa, özellikle uzun süre çalışan uygulamalarda biriken ciddi bellek sızıntılarına yol açabilir ve kullanılabilir GPU belleğini önemli ölçüde azaltabilir. Bu nedenle bellek sızıntılarını önlemek için cudaDeviceReset() yerine doğru zamanda cudaFree() kullanımı tercih edilmelidir. cuda-memcheck aracı ile kodu çalıştırdıktan sonra ekranda gösterilen bilgileri tek tek inceleyelim:

```
CUDA-MEMCHECK
Leaked 12 bytes at 0x7047a0000
LEAK SUMMARY: 12 bytes leaked in 1 allocations
ERROR SUMMARY: 0 errors
```

- **Leaked 12 bytes at 0x7047a0000:** GPU üzerinde 12 baytlık bir bellek alanı ayrılmış, ancak program bitene kadar cudaFree() ile serbest bırakılmamış. 0x7047a0000 değeri sızıntının başladığı bellek bölgesinin adresidir.
- **LEAK SUMMARY: 12 bytes leaked in 1 allocations:** Programda bir adet bellek ayırma işlemi serbest bırakılmamış ve bu nedenle toplam 12 baytlık bellek sızıntısı meydana gelmiştir.
- **ERROR SUMMARY: 0 errors:** Bellek erişim hatası oluşmadığını bildirmektedir.

1.1.2 Initcheck: CUDA uygulamalarında başlatılmamış (uninitialized) GPU ana bellek bölgelerine yapılan erişimleri çalışma zamanı (run time) sırasında tespit eden bir hata ayıklama aracıdır. Initcheck aracı, GPU ana belleğinin başlatılmamış olup olmadığını aşağıdaki yollarla tespit edebilir:

- Aygıt tarafındaki (device side) yazma işlemleri
- CUDA memcpy API çağrıları

- CUDA memset API çağrıları

Initcheck yalnızca GPU ana belleğine (device global memory) yapılan erişimleri tespit edebilir; ortak bellek (shared memory) veya yerel bellek (local memory) erişimlerini kapsamaz. Initcheck'in kullanılabilmesi için programın --tool initcheck seçeneği ile çalıştırılması gerekir. Initcheck kullanılmadan önce, uygulamanın bellek erişim hatalarından (memory access errors) arındırılmış olduğundan emin olmak için cuda-memcheck aracının çalıştırılması önerilir. Aşağıdaki örnekte GPU ana belleğinde 32 bayt yer ayrılmakta ve sekiz tane iş parçacığına (thread) sahip olan kernel içinde her bir thread kendi thread numarasını kullanarak 4 baytlık bellek bölgesine erişmeye çalışmaktadır. Öncesinde cudaMemset komutu ile ayrılmış bellek bölgesindeki tüm elemanların değeri 0 yapılmaya çalışılmaktadır. cudaMemset komutunda üçüncü parametrede toplam eleman sayısı yerine elemanların toplam kapladığı alanı bayt cinsinden vermek gerekir. Bu örnekte 8 değeri girilerek 8 baytlık bir bölgeye 0 değeri atansın denilmektedir. Bu durumda sadece ilk iki elemanın değeri 0 olmakta diğerleri hala uninitialized olarak gözükmektedir ve ortaya memset hatası çıkmaktadır. Bu örneğin CUDA kodu aşağıda verilmektedir:

```
__global__ void memset_error(int *A)
{
    A[threadIdx.x] += threadIdx.x;
}

int main()
{
    int *A_GPU;
    cudaMalloc((void**)&A_GPU,sizeof(int)*8);
    cudaMemset(A_GPU,0,8);
    memset_error<<<1,8>>>(A_GPU);
    cudaDeviceSynchronize();
    cudaFree(A_GPU);
}
```

A_GPU ile işaret edilen bellek bölgesi başlatılırken hata oluştu! Üçüncü parametre bayt cinsinden verilmeli.

Bu kodu aşağıdaki gibi derleyip çalıştırabiliriz:

$ nvcc -o initcheck1 initcheck1.cu

$ cuda-memcheck --tool initcheck initcheck1

cuda-memcheck ile çalıştırdıktan sonra ekranda gözüken bilgiler tek tek inceleyelim:

```
CUDA-MEMCHECK
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (7,0,0) in block (0,0,0)
    Address 0x7047a001c
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (6,0,0) in block (0,0,0)
    Address 0x7047a0018
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (5,0,0) in block (0,0,0)
    Address 0x7047a0014
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (4,0,0) in block (0,0,0)
    Address 0x7047a0010
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (3,0,0) in block (0,0,0)
    Address 0x7047a000c
Uninitialized __global__ memory read of size 4
    at 0x00000030 in memset_error(int*)
    by thread (2,0,0) in block (0,0,0)
    Address 0x7047a0008
ERROR SUMMARY: 6 errors
```

- **Uninitialized _global_ memory read of size 4:** Başlatılmamış GPU ana bellek bölgesinden 4 baytlık okuma yapıldığını bildirmektedir.
- **at 0x00000030 in memset_error(int*):** Hatanın oluştuğu kernel'in ismini ve hataya sebep veren komutun kernel içindeki konumunu talimat adresi (instruction offset) olarak göstermektedir.
- **by thread (7,0,0) in block (0,0,0):** Hatanın hangi thread ve hangi block tarafından yapıldığını belirtmektedir. Thread numarası 2,3,4,5,6,7 olan thread'ler memset hatası ile karşılaşmışlardır.
- **Address 0x7047a001c:** Thread numarası 7 olan thread tarafından erişilmeye çalışılan başlatılmamış (uninitialized) bellek bölgesinin adresini göstermektedir. Diğer thread'lerde de bu mesaj aynı anlama gelmektedir.
- **ERROR SUMMARY: 6 errors:** Thread numarası 0 ve 1 olanlar başlatılmış (initialized) bellek bölgelerine erişirken diğer 6 thread başlatılmamış (uninitialized) bellek bölgelerine erişmeye çalışmaktadır. Bu yüzden 6 tane hata oluşmuştur.

1.2. CUDA-GDB:

CUDA-GDB, Linux ve QNX işletim sistemlerinde CUDA uygulamalarının hatalarının ayıklanması (debugging) için kullanılan NVIDIA tarafından geliştirilmiş bir araçtır. CUDA-GDB, GNU GDB'nin (GNU Debugger) CUDA'ya özel olarak genişletilmiş (extension) bir sürümüdür. CUDA-GDB:

- C / C++ ve Fortran ile yazılmış CUDA uygulamalarının hata ayıklanmasını destekler.
- Kullanıcılara aşağıdaki olanakları sağlar:
    a. Breakpoint (kesme noktası) tanımlama
    b. CUDA uygulamalarını adım adım çalıştırma (single-step)
    c. Belleği (memory) inceleme ve değiştirme
    d. GPU üzerinde çalışan herhangi bir thread'e ait değişkenleri inceleme ve değiştirme
- CUDA kernel'lerinin ve GPU üzerinde çalışan iş parçacıklarının detaylı olarak incelenmesine imkân tanır ve paralel çalışan CUDA uygulamalarında hata ayıklamayı mümkün kılar.

1.3. Nsight:

NVIDIA Nsight, CUDA uygulamaları için görsel (visual) hata ayıklama imkânı sunan NVIDIA araç ailesidir. Grafiksel arayüz sayesinde CUDA uygulamalarının çalışma süreci görsel olarak izlenebilir ve analiz edilebilir. Bazı araçlar aşağıda verilmektedir:

- **NVIDIA Nsight Visual Studio Edition:**
    - Microsoft Visual Studio ortamına entegre çalışan bir CUDA hata ayıklama aracıdır.
    - CUDA kernel'lerinin, thread'lerin ve bellek erişimlerinin görsel olarak incelenmesini sağlar.
    - Özellikle Windows üzerinde CUDA geliştirme ve hata ayıklama süreçlerinde yaygın olarak kullanılır.
    - Nsight Visual Studio Edition hâlen kullanılmakta ve güncel CUDA Toolkit sürümleriyle çalışmaktadır; ancak desteği zamanla güncellenmekte ve bazı eski platformlar destek dışı bırakılmaktadır.
- **NVIDIA Nsight Eclipse Edition:**
    - Eclipse IDE ile entegre çalışan bir CUDA geliştirme ve hata ayıklama aracıdır.
    - Linux tabanlı sistemlerde CUDA kernel'lerinin görsel olarak debug edilmesine olanak tanır.

- CUDA uygulamalarının çalışması sırasında thread, bellek ve kernel davranışları ayrıntılı biçimde analiz edilebilir.
- Nsight Eclipse Edition'in eski bağımsız sürümü artık sunulmamaktadır; modern CUDA Toolkit'te benzer işlevler Eclipse eklentileri (plugins) şeklinde sağlanmaktadır.

2. CUDA Profiling:

Profiling araçları, CUDA uygulamalarının performansını anlamak, analiz etmek ve optimize etmek için kullanılır. Bu araçlar sayesinde uygulamanın GPU üzerinde nasıl çalıştığı ayrıntılı olarak incelenebilir ve performans darboğazları (bottlenecks) tespit edilebilir. Bazı profiling araçları aşağıda verilmektedir:

- **Visual Profiler:**
  - CUDA uygulamalarının performansını görselleştirmeye ve optimize etmeye olanak tanır.
  - Uygulamanın çalışma sürecini gösteren zaman çizelgesi (timeline) sunar.
  - Kernel çalışmaları, bellek kopyalamaları ve GPU aktiviteleri ayrıntılı olarak izlenebilir.
  - Uygulamayı analiz ederek olası performans darboğazlarını tespit eder.
  - Bu aracın bağımsız (standalone) sürümü nvvp olarak adlandırılır.
- **Nvprof:**
  - CUDA uygulamalarına ait profiling verilerini komut satırından (command-line) toplamaya ve görüntülemeye imkân tanır.
  - Kernel çalışma süreleri, bellek erişimleri ve API çağrıları gibi performans metriklerini raporlar.
  - Grafik arayüz gerektirmeden, özellikle uzak sistemlerde (remote systems) ve otomatik analizlerde kullanışlıdır.

2.1. NVPROF:

Bu araç CUDA uygulamalarına ait profiling verilerinin komut satırı (command-line) üzerinden toplanmasını ve görüntülenmesini sağlar. Aşağıdaki CUDA ile ilişkili aktivitelerin zaman çizelgesini (timeline) hem CPU hem de GPU tarafında toplar ve analiz eder:

- Kernel çalıştırmaları (kernel executions)
- Bellek aktarımları (memory transfers)
- Bellek sıfırlama işlemleri (memory set) ve CUDA API çağrıları
- CUDA kernel'larına ait olaylar (events) veya performans metrikleri (metrics)

Bu sayede uygulamanın CPU–GPU etkileşimi, kernel çalışma davranışı ve bellek erişim performansı ayrıntılı olarak incelenebilir. nvprof komutunun formatı şu şekildedir:

$ nvprof [options] [application] [application-arguments]

Bazı options bilgileri aşağıda verilmektedir:

- **query-events:** GPU üzerinde kullanılabilir olan tüm olayları (events) listeler.
- **query-metrics:** GPU üzerinde kullanılabilir olan tüm performans metriklerini (metrics) listeler.

- **print-api-summary:** CUDA runtime ve driver API çağrılarının özetini ekrana yazdırır.
- **print-api-trace:** CUDA runtime ve driver API çağrılarının ayrıntılı izini (trace) gösterir.
- **print-gpu-summary:** GPU üzerinde gerçekleşen aktivitelerin özetini yazdırır (CUDA kernel ve memcpy/memset işlemleri dâhil).
- **print-gpu-trace:** Tek tek kernel çağrılarını (memcpy/memset işlemleri dâhil) zaman sırasına göre listeler. Event/metric profiling modunda her kernel çağrısına ait event ve metric bilgilerini de gösterir.
- **log-file:** nvprof çıktılarının tamamını belirtilen dosyaya veya standart çıktı kanallarından birine yönlendirir.

CUDA profiling araçları, uygulamanın performansını farklı bakış açılarından analiz edebilmek için çeşitli profiling modları sunar:

- **Summary Modu:**
    - Varsayılan (default) çalışma modudur.
    - Her bir CUDA kernel fonksiyonu ve her bir CUDA bellek kopyalama/sıfırlama (memory copy/set) işlemi için tek satırlık özet sonuç üretir.
    - Uygulamanın genel performans durumu hakkında hızlı bir bakış sağlar.
- **GPU-Trace ve API-Trace Modları:**
    - GPU-Trace modu GPU üzerinde gerçekleşen tüm aktivitelerin zaman çizelgesini (timeline) kronolojik sırayla sunar. Kernel çalışmaları ve bellek işlemleri ayrıntılı olarak izlenebilir.
    - API-Trace modu CPU tarafında çağrılan tüm CUDA runtime ve driver API çağrılarının zaman çizelgesini kronolojik sırayla gösterir. CPU–GPU etkileşiminin analiz edilmesine olanak tanır.
- **Event/Metric Modu:**
    - Event/Metric Summary modunda event veya metric değerleri, tüm kernel çalıştırmaları boyunca toplu olarak ölçülür ve raporlanır. Genel performans eğilimlerinin analiz edilmesinde kullanılır.
    - Event/Metric Trace modunda her bir kernel çalışması için ayrı ayrı event ve metric değerlerini gösterir. Kernel bazlı ayrıntılı performans analizi yapılmasına imkân tanır.

Bu modları incelemek için aşağıda kodu verilen uygulamayı kullanacağız. Bu uygulamada iki tane tek boyutlu dizi üzerinde vektör toplama ve vektör çıkarma işlemleri yapılmaktadır. Dizilerin eleman sayısı 32768 olarak alınmıştır. Bir bloktaki thread sayısı da 1024 olarak belirlenmiştir.

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

__global__ void vector_sub(float *A,float *B,float *C)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    C[tid] = A[tid] - B[tid];//Vector subtraction
}

int main(int argc, char **argv)
{
    int size = 32768;//Data size
    float *A_host = new float[size];//Host Array
    float *B_host = new float[size];//Host Array
    float *C_host = new float[size];//Host Array

    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
    cudaMalloc((void**)&A_GPU,sizeof(float)*size);
    cudaMalloc((void**)&B_GPU,sizeof(float)*size);
    cudaMalloc((void**)&C_GPU,sizeof(float)*size);

    dim3 threadsPerBlock(1024);//Number of threads in a block
    dim3 numBlocks(size/1024);//Number of blocks in a grid

    for(int counter = 0;counter < 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)*size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*size,cudaMemcpyHostToDevice);

    vector_add<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
    cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);

    vector_sub<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);
    cudaMemcpy(C_host,C_GPU,sizeof(float)*size,cudaMemcpyDeviceToHost);

    delete[] A_host;
    delete[] B_host;
    delete[] C_host;
    cudaFree(A_GPU);
    cudaFree(B_GPU);
    cudaFree(C_GPU);
}
```

2.1.1 Summary Modu: Kodu summary modunda aşağıdaki gibi derleyip çalıştırabilirsiniz:

$ nvcc -o vector_add_sub vector_add_sub.cu

$ nvprof ./vector_add_sub

Ekran çıktısı şekildedir:

```
==22396== NVPROF is profiling process 22396, command: ./vector_add_sub
==22396== Profiling application: ./vector_add_sub
==22396== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.64%  31.552us         2  15.776us  15.424us  16.128us  [CUDA memcpy HtoD]
 43.64%  30.848us         2  15.424us  15.264us  15.584us  [CUDA memcpy DtoH]
  6.02%  4.2560us         1  4.2560us  4.2560us  4.2560us  vector_add(float*, float*, float*)
  5.70%  4.0320us         1  4.0320us  4.0320us  4.0320us  vector_sub(float*, float*, float*)

==22396== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.63%  80.621ms         3  26.874ms  2.1980us  80.616ms  cudaMalloc
  0.72%  585.52us       166  3.5270us      74ns  139.78us  cuDeviceGetAttribute
  0.30%  243.76us         4  60.939us  39.615us  90.819us  cudaMemcpy
  0.15%  123.96us         3  41.320us  3.3530us  111.96us  cudaFree
  0.09%  73.572us         2  36.786us  36.607us  36.965us  cuDeviceTotalMem
  0.07%  55.062us         2  27.531us  26.456us  28.606us  cuDeviceGetName
  0.04%  34.634us         2  17.317us  17.079us  17.555us  cudaLaunch
  0.00%  2.9150us         6     485ns      91ns  2.1350us  cudaSetupArgument
  0.00%  1.7640us         2     882ns     291ns  1.4730us  cuDeviceGetCount
  0.00%  1.2110us         2     605ns     264ns     947ns  cudaConfigureCall
  0.00%     917ns         4     229ns      94ns     530ns  cuDeviceGet
```

Tabloda sütunlarda verilen bilgilerin açıklamaları şu şekildedir:

- **Time(%):** İlgili işlemin süresinin toplam GPU süresine olan oranın yüzdesel gösterimi
- **Time:** İlgili işlemin harcadığı toplam süre
- **Calls:** İlgili işlemin kaç kez çağrıldığı
- **Avg:** İlgili işlemin harcadığı ortalama süre
- **Min / Max:** İlgili işlemin harcadığı en küçük ve en büyük süreler
- **Name:** İlgili işlemin adı

Tablonun üst kısmında GPU aktivite özeti (GPU summary) yani GPU tarafından gerçekleşen aktivitelerin özet profil sonuçları gösterilmektedir. Bu aktiviteler kernel çalışmalarını ve GPU bellek işlemlerini (cudaMemcpy, cudaMemset gibi) içermektedir. Tablo sonuçlarını maddeler halinde yorumlayalım:

- CPU'dan GPU'ya ve GPU'dan CPU'ya doğru bellek transferleri (CUDA memcpy) gerçekleştirilmiştir. Her bir yön için bu transfer işlemi iki kez gerçekleşmiştir.
- CPU'dan GPU'ya olan bellek transferlerinin toplam süresi 31.552us'dir. Toplam harcanan GPU süresinin %44.64'ünü oluşturmaktadır. Bu iki işlemin harcanan sürelerinin ortalaması ise 15.776us'dir.
- GPU'dan CPU'ya olan bellek transferlerinin toplam süresi 30.848us'dir. Toplam harcanan GPU süresinin %43.64'ünü oluşturmaktadır. Bu iki işlemin harcanan sürelerinin ortalaması ise 15.424us'dir.
- vector_add ve vector_sub isimli iki tane CUDA kernel birer kez çağrılmıştır. vector_add kernel işleminin toplam süresi 4.256us'dir. Toplam harcanan GPU süresinin %6.02'isini oluşturmaktadır. vector_sub kernel işleminin toplam süresi 4.032us'dir. Toplam harcanan GPU süresinin %5.70'ini oluşturmaktadır.
- Toplam sürenin yaklaşık %88'i bellek transfer işlemlerine harcanmaktadır. Dolayısıyla bu uygulama hesaplama ağırlıklı değil, bellek transferi ağırlıklı bir uygulamadır.

Tablonun alt kısmında ise CPU tarafında çağrılan CUDA API fonksiyonlarının özet profil sonuçları gösterilmektedir. Bazı öne çıkan API çağrıları cudaMalloc, cuDeviceGetAttribute, cudaMemcpy ve cudaFree API çağrılarıdır. Tablo sonuçlarını maddeler halinde yorumlayalım:

- cudaMalloc üç kez çağrılmıştır. Toplam harcanan süre 80.621ms olarak ölçülmüştür. Bu süre toplam harcanan API çağırma sürelerinin %98.63'üne denk gelmektedir. Bu üç çağırma içinden en uzun süren çağrı 80.616ms sürmüştür. En kısa süren ise 2.198us sürmüştür.
- cuDeviceGetAttribute 166 kez çağrılmıştır. Toplanan harcanan süre 585.52us'dir. Ortalama olarak 3.527us zaman harcamış olsa da çok fazla kez çağrıldığı için toplam harcanan süre olarak diğer birçok API'nın önünde çıkmıştır.
- cudaMemcpy dört kez çağrılmıştır. Toplam harcanan süre 243.76us'dir. En uzun süren çağrı 90.819us, en kısa süren çağrı 39.615us sürmüştür. Not: Tablonun üst kısmındaki CUDA memcpy ile ilgili sonuçlar transfer işleminin gerçekleşme maliyetini verirken buradaki cudaMemcpy işlemi ile ilgili sonuçlar API fonksiyon çağrısının maliyetini vermektedir.
- cudaFree üç kez çağrılmıştır. Toplam harcanan süre 123.96us olarak ölçülmüştür. Bu üç çağırma içinden en uzun süren çağrı 111.96us sürmüştür. En kısa süren ise 3.353us sürmüştür.
- cudaMalloc, toplam API süresinin neredeyse tamamını (%98.6) oluşturmaktadır. Bellek ayırma işlemleri çok pahalıdır ve uygulama süresini ciddi şekilde etkiler. CUDA

uygulamalarında bellek ayırma işlemleri mümkün olduğunca azaltılmalı, yeni bir bellek işlemi yapılacağında daha önce yer ayırma işlemi yapılmış ve artık kullanılmayan bellek bölgeleri yeniden yer ayırma işlemi yapılmadan tekrar kullanılmalıdır.

Not: Sadece tablonun üst kısmındaki verileri göstermek istiyorsak şu şekilde çalıştırmalıyız:

$ nvprof --print-gpu-summary ./vector_add_sub

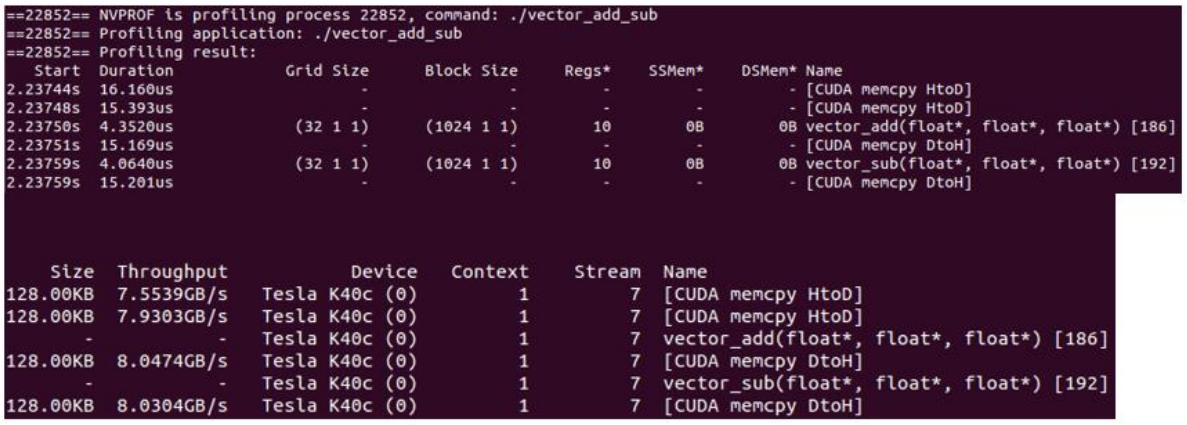
Sadece tablonun alt kısmındaki verileri göstermek istiyorsak şu şekilde çalıştırmalıyız:

$ nvprof --print-api-summary ./vector_add_sub

2.1.2 GPU-Trace ve API-Trace Modları: Kodu aşağıdaki gibi bir komutla GPU-Trace modunda çalıştırabilirsiniz:

$ nvprof --print-gpu-trace ./vector_add_sub

Elde edilen çıktılar aşağıda verilmektedir. Ayrıca her bir çıktının ne anlama geldiği maddeler halinde anlatılmaktadır:



```
==22852== NVPROF is profiling process 22852, command: ./vector_add_sub
==22852== Profiling application: ./vector_add_sub
==22852== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem* Name
2.23744s  16.160us                    -               -         -         -         - [CUDA memcpy HtoD]
2.23748s  15.393us                    -               -         -         -         - [CUDA memcpy HtoD]
2.23750s  4.3520us              (32 1 1)      (1024 1 1)        10        0B        0B vector_add(float*, float*, float*) [186]
2.23751s  15.169us                    -               -         -         -         - [CUDA memcpy DtoH]
2.23759s  4.0640us              (32 1 1)      (1024 1 1)        10        0B        0B vector_sub(float*, float*, float*) [192]
2.23759s  15.201us                    -               -         -         -         - [CUDA memcpy DtoH]



    Size  Throughput           Device   Context    Stream  Name
128.00KB  7.5539GB/s   Tesla K40c (0)         1         7  [CUDA memcpy HtoD]
128.00KB  7.9303GB/s   Tesla K40c (0)         1         7  [CUDA memcpy HtoD]
       -           -   Tesla K40c (0)         1         7  vector_add(float*, float*, float*) [186]
128.00KB  8.0474GB/s   Tesla K40c (0)         1         7  [CUDA memcpy DtoH]
       -           -   Tesla K40c (0)         1         7  vector_sub(float*, float*, float*) [192]
128.00KB  8.0304GB/s   Tesla K40c (0)         1         7  [CUDA memcpy DtoH]
```

- **Start:** İşlemin başlama zamanıdır. Profiling başlangıcından itibaren geçen süreyi saniye cinsinden gösterir.
- **Duration:** İşlemin ne kadar sürdüğünü gösterir. Kernel çalışması veya bellek transfer işleminin GPU'da harcadığı süredir.
- **Grid Size:** Kernel çağrılarında kullanılan grid boyutunu belirtir. Bu örnekteki kernel'lerde 32 tane blok oluşturulduğu bildirilmektedir. Bellek transfer işlemlerinde ise geçerli değildir.
- **Block Size:** Kernel çağrılarında kullanılan blok boyutunu gösterir. Bu örnekte oluşturulan her bir bloğun 1024 tane iş parçacığına (thread) sahip olduğu bildirilmektedir. Bellek transfer işlemlerinde ise geçerli değildir.
- **Regs*:** Kernel başına her thread'in kullandığı register sayısını gösterir. Register kullanımı, occupancy ve performansı doğrudan etkiler.
- **SSMem*:** Kernel tarafından kullanılan statik shared memory miktarını gösterir.
- **DSMem*:** Kernel tarafından kullanılan dinamik shared memory miktarını gösterir.
- **Size:** Transfer edilen verilerin büyüklüğünü bayt cinsinden gösterir.
- **Throughput:** Bellek transferinin etkin bant genişliğini gösterir. GB/s cinsindendir. Yüksek değerler daha verimli veri transferini ifade eder.
- **Device:** İşlemin çalıştığı GPU cihazını belirtir.

- **Context:** CUDA işleminin ait olduğu CUDA context numarasıdır. Aynı GPU üzerinde birden fazla context olabilir.
- **Stream:** İşlemin yürütüldüğü CUDA stream numarasını gösterir. Aynı stream içindeki işlemler sıralı çalışır.
- **Name:** Gerçekleşen işlemin adıdır.

Kodu aşağıdaki gibi bir komutla API-Trace modunda da çalıştırabilirsiniz:

$ nvprof --print-api-trace ./vector_add_sub

Elde edilen çıktılar aşağıda verilmektedir. Ayrıca her bir çıktının ne anlama geldiği maddeler halinde anlatılmaktadır:

```
==22887== NVPROF is profiling process 22887, command: ./vector_add_sub
==22887== Profiling application: ./vector_add_sub
==22887== Profiling result:
   Start  Duration  Name
2.32905s  1.3060us  cuDeviceGetCount
2.32906s     234ns  cuDeviceGet
2.32916s     805ns  cuDeviceGet
2.32932s     273ns  cuDeviceGetCount
2.32932s     105ns  cuDeviceGet
2.32932s  31.157us  cuDeviceGetName
2.32935s  38.126us  cuDeviceTotalMem
2.32939s     150ns  cuDeviceGetAttribute
2.32939s      84ns  cuDeviceGetAttribute
2.32939s     100ns  cuDeviceGetAttribute
2.32939s     101ns  cuDeviceGetAttribute
2.32939s      80ns  cuDeviceGetAttribute
2.32939s  25.346us  cuDeviceGetAttribute
2.32942s     103ns  cuDeviceGetAttribute
2.32942s      82ns  cuDeviceGetAttribute
2.32942s      81ns  cuDeviceGetAttribute
2.32942s      88ns  cuDeviceGetAttribute
2.32942s      90ns  cuDeviceGetAttribute
<...more output...>
```

- **Start:** CUDA API çağrısının başladığı zamanı gösterir. Profiling başlangıcından itibaren geçen süreyi saniye cinsinden ifade eder.
- **Duration:** İlgili CUDA API çağrısının CPU tarafında ne kadar sürdüğünü gösterir. Bu süre, GPU kernel çalışma süresi değildir. Sürücü(driver)/çalışma zamanı(runtime) tarafından harcanan zamandır.
- **Name:** Çağrılan CUDA Runtime veya Driver API fonksiyonunun adıdır.

2.1.3 Event/Metric Modu: Event ve metric ile ilgili bazı açıklamalar şu şekildedir:

- **Event (Olay):** GPU (device) üzerinde gerçekleşen, sayılabilir (countable) bir etkinlik, işlem veya olaydır ve kernel çalışması (kernel execution) sırasında toplanır.
- **Metric (Metrik):** Bir veya birden fazla event (olay) kullanılarak hesaplanan, uygulamaya ait istatistiksel bir değerdir.
- Mevcut event (olay) listesini görüntülemek için **nvprof --query-events** komutunu çalıştırın.
- Mevcut metric (metrik) listesini görüntülemek için **nvprof --query-metrics** komutunu çalıştırın.

Örneğimizi Event/Metric Summary modunda aşağıdaki gibi bir komutla çalıştırabilirsiniz:

$ nvprof --events warps_launched --metrics gld_transactions ./vector_add_sub

Bu komutta event olarak warps_launched (çalışan warp sayısı) metrik olarak gld_transactions (global bellekten yapılan okuma (load) işlemlerinin toplam sayısı) kullanılmaktadır. Sonuçlar aşağıda verilmektedir:

```
==23567== Profiling result:
==23567== Event result:
Invocations                                Event Name         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: vector_add(float*, float*, float*)
          1                            warps_launched        1024        1024        1024
        Kernel: vector_sub(float*, float*, float*)
          1                            warps_launched        1024        1024        1024

==23567== Metric result:
Invocations                              Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: vector_add(float*, float*, float*)
          1                          gld_transactions                Global Load Transactions        2048        2048        2048
        Kernel: vector_sub(float*, float*, float*)
          1                          gld_transactions                Global Load Transactions        2048        2048        2048
```

- **Event Sonuçları – warps_launched:**
  - Her iki kernel için de 1024 warp başlatıldığı görülmektedir. Bu değer kernel'in çalıştırdığı thread sayısının GPU üzerinde kaç warp'a bölündüğünü göstermektedir.
- **Metric Sonuçları – gld_transactions:**
  - Her iki kernel için 2048 adet global bellek okuma işlemi (global load transaction) gerçekleşmiştir. A ve B değişkenlerinin bellekte işaret ettiği bölgeler okunmaktadır.
  - Warp içindeki thread'ler global belleğe coalesced (birleştirilmiş) şekilde eriştiği için her warp mümkün olan en az sayıda bellek işlemi üretmiştir. Yani bir warp'taki 32 thread okuma işlemlerini tek işlemde (transaction) gerçekleştirmektedir. Böylelikle 1024 warp iki ayrı değişken üzerinden toplamda 2048 adet global bellek okuma işlemi yapmaktadır.
  - Eğer bellek erişimleri coalesced olmasaydı gld_transactions değeri daha büyük olurdu. Bu durum bellek bant genişliğinin daha verimsiz kullanılmasına yol açardı.
- **Event/Metric Trace modu:**
  - Summary modunda aggregate-mode varsayılan olarak açıktır (on). Bu nedenle SM (Streaming Multiprocessor) bazlı event ve metrikler, GPU üzerindeki tüm SM'ler için birleştirilerek (aggregate) tek bir sonuç olarak gösterilir.
  - Trace modunda ise aggregate-mode kapalı (off) olarak kullanılır. Bu durumda her bir SM'ye ait event ve metrik değerleri ayrı ayrı raporlanır.

Bazı performans metrikleri aşağıda verilmektedir:

- **l1_cache_global_hit_rate:** Global bellekten yapılan okumalarda L1 önbellek (L1 cache) isabet oranını (hit rate) ifade eder.
- **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.
- **sm_efficiency:** GPU üzerindeki tüm çoklu işlemciler (Streaming Multiprocessors – SM) için ortalama alınarak, bir SM üzerinde en az bir warp'ın aktif olduğu sürenin yüzdesini gösterir.
- **achieved_occupancy:** Bir SM üzerinde, aktif döngüler (cycle) başına ortalama aktif warp sayısının, o SM'in desteklediği maksimum warp sayısına oranını ifade eder.

- **warp_execution_efficiency:** Bir warp içindeki ortalama aktif thread sayısının, bir warp tarafından desteklenen maksimum thread sayısına oranını yüzde cinsinden gösterir.

2.2. Diğer CUDA Profiling Araçları:

Yeni CUDA sürümlerinde nvprof artık kullanılmıyor (deprecated). Onun yerine Nsight tabanlı iki yeni araç kullanılıyor:

- **NVIDIA Nsight Systems:** Sistem seviyesinde profiling için nvprof'un zaman çizelgesi (timeline) ve genel performans analizine karşılık gelir. Şunları analiz eder:
    - Kernel çalışmaları
    - CPU–GPU etkileşimi
    - CUDA API çağrıları
    - Memory kopyalamaları
- **NVIDIA Nsight Compute:** Kernel seviyesinde detaylı profiling için tek tek kernel'leri ayrıntılı şekilde analiz eder. nvprof --metrics kullanımının yerini alır. Analiz ettiği bazı metrikler şunlardır:
    - Occupancy
    - Memory throughput
    - Warp verimliliği
    - Instruction istatistikleri