CUDA Programlamaya Giriş

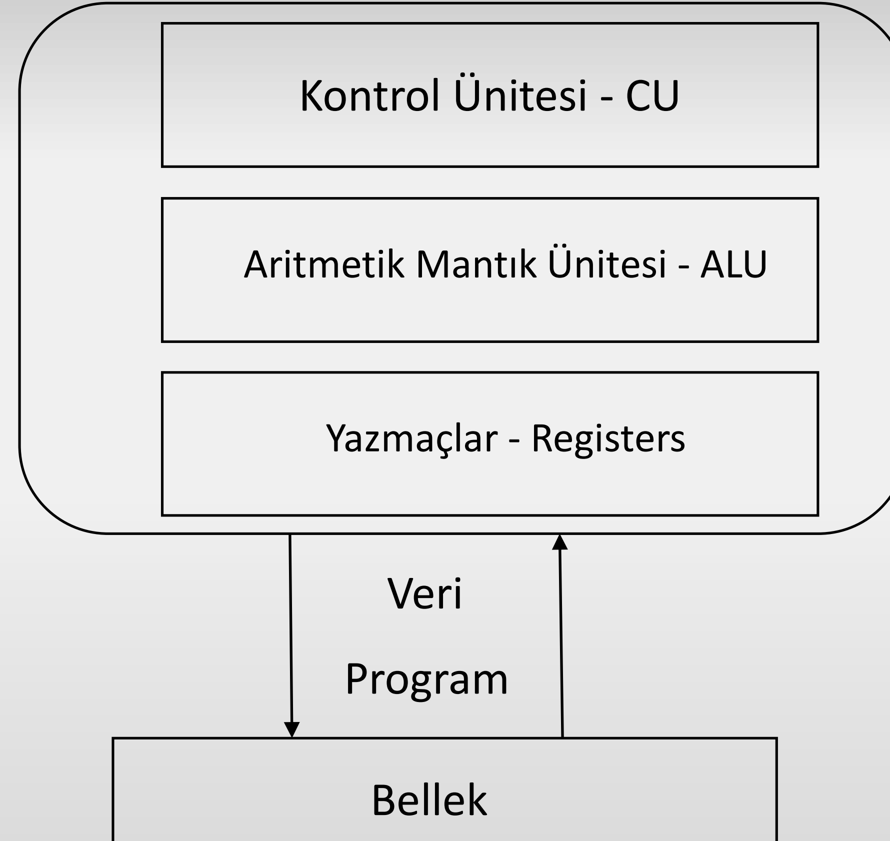
Dr. Özcan Dülger
Bilgisayar Mühendisliği, Orta Doğu Teknik Üniversitesi
Bilgisayar Mühendisliği, Artvin Çoruh Üniversitesi

EURO

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

Von Neumann Mimarisi



Von Neumann Mimarisi

Merkezi İşlem Birimi - CPU

Kontrol Ünitesi - CU → Program komutları işleniyor

Aritmetik Mantık Ünitesi - ALU → Aritmetik ve mantıksal işlemler yapılıyor

Yazmaçlar - Registers → Geçici veriler saklanıyor

Veri

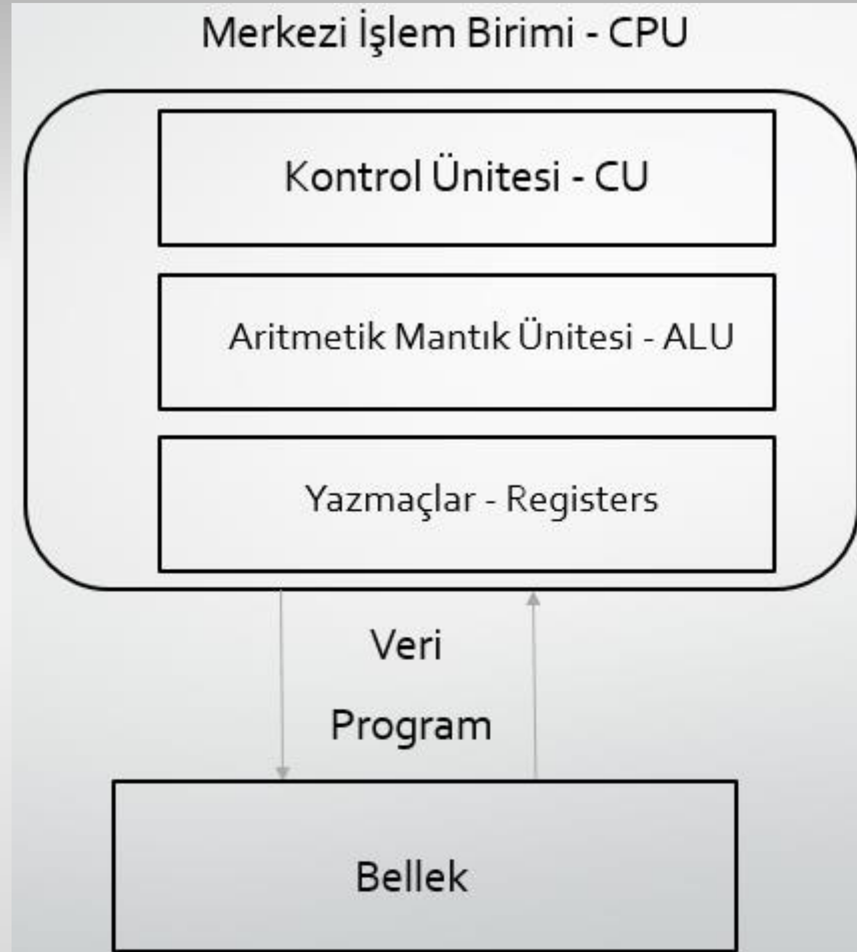
Program

Bellek → Veriler ve program komutları saklanıyor

CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

Von Neumann Mimarisi



Program 1

x = 5 → Komut1

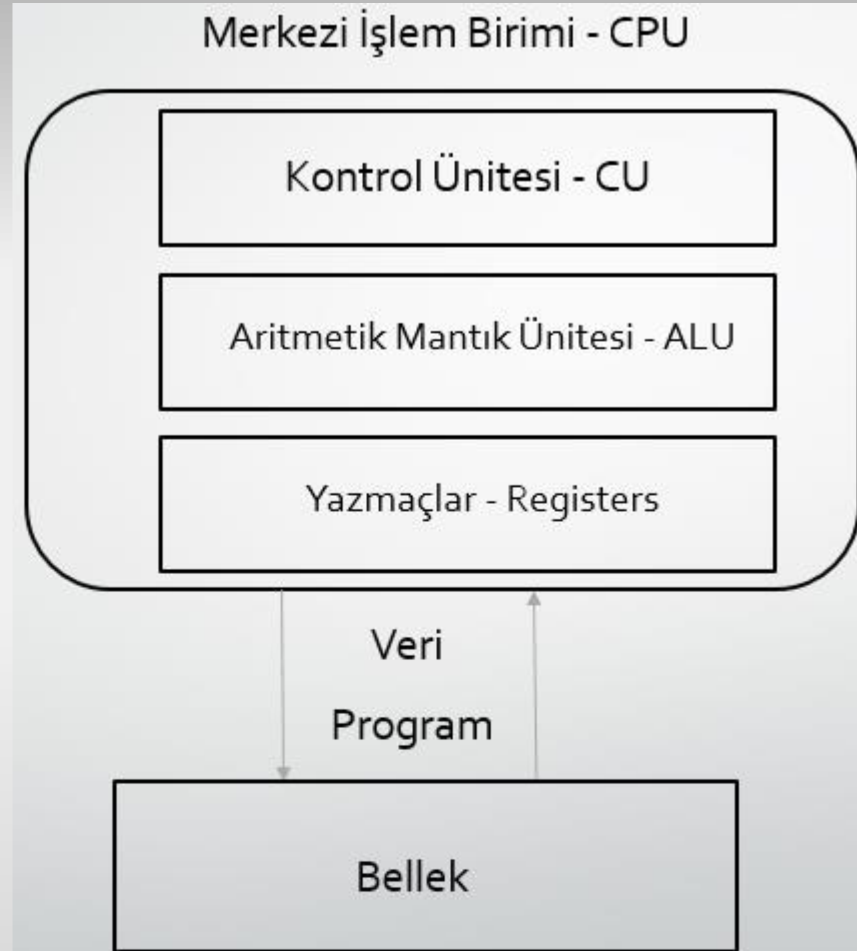
y = x * 3 + 2 → Komut2

*

*

*

Von Neumann Mimarisi



Program 1 – Komut 1
Program 2 – Komut 1
Program 3 - Komut 1
Program 2 – Komut 2
Program 1 – Komut 2
Program 4 – Komut1
Program 3 – Komut 2
Program 1 – Komut 3
*
*
*

Çok Çekirdekli İşlemciler

Merkezi İşlem Birimi - CPU

Kontrol Ünitesi - CU

Aritmetik Mantık Ünitesi - ALU

Yazmaçlar - Registers

Merkezi İşlem Birimi - CPU

Kontrol Ünitesi - CU

Aritmetik Mantık Ünitesi - ALU

Yazmaçlar - Registers

Veri

Program

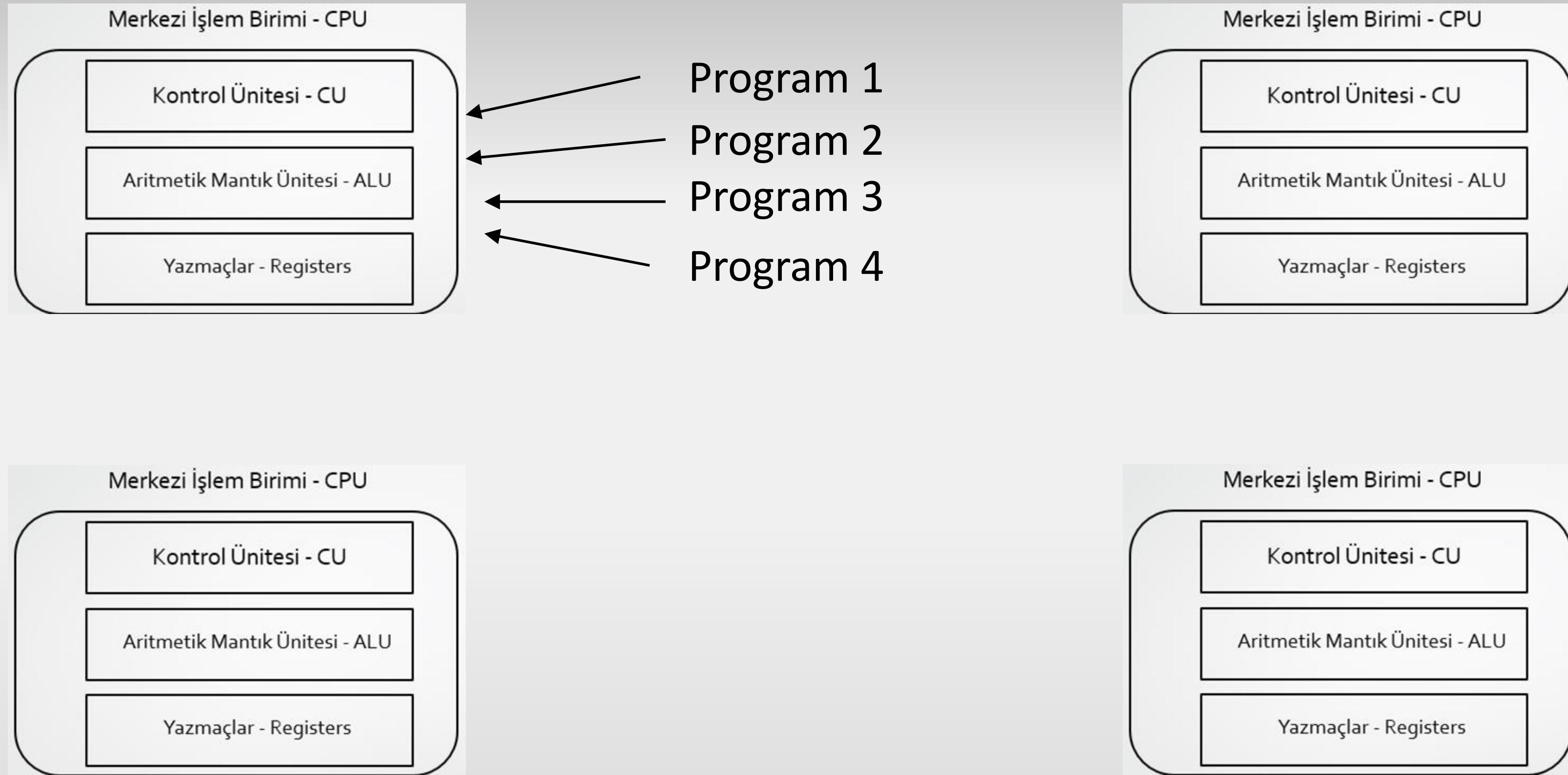
Veri

Program

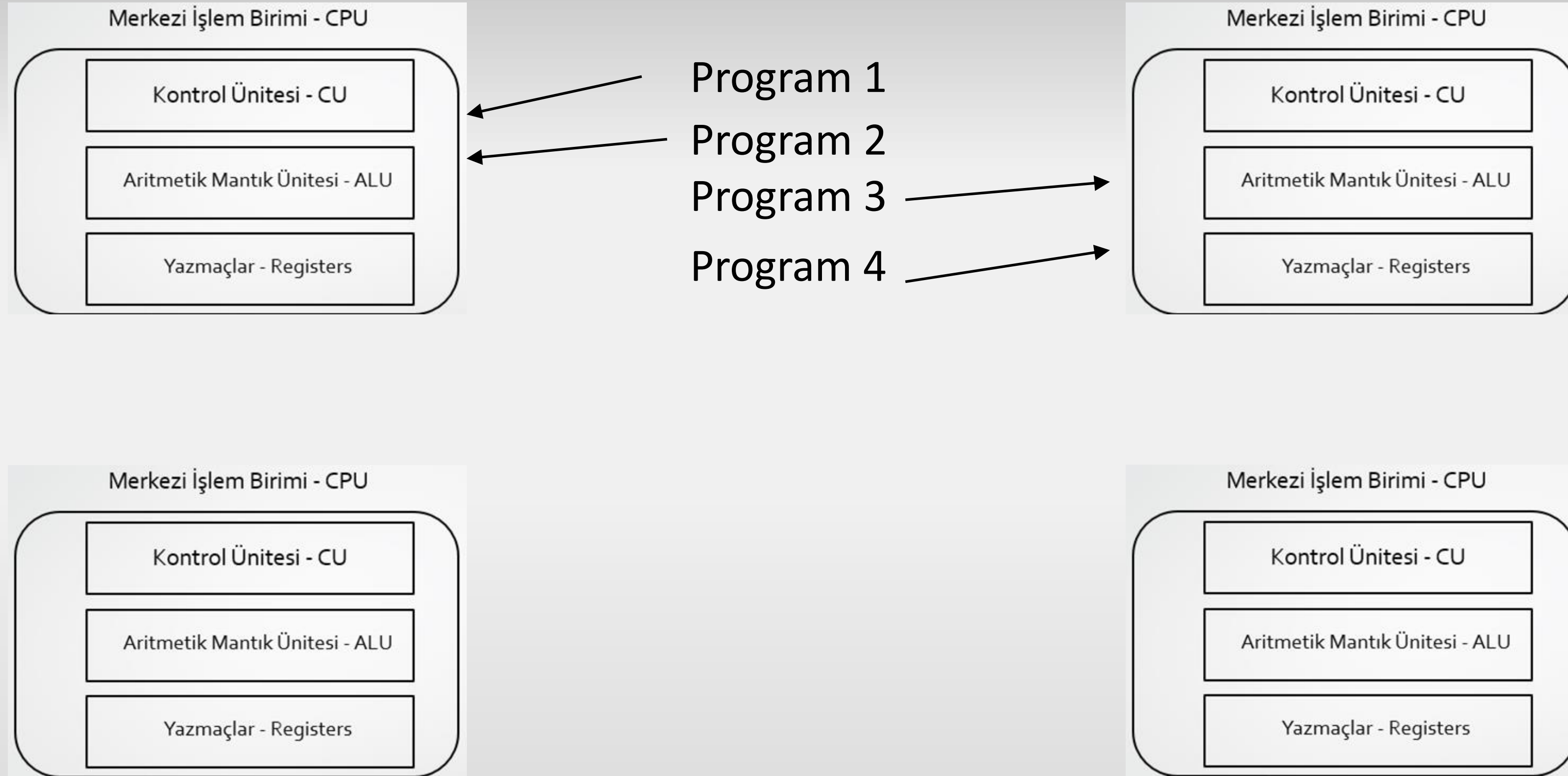
Bellek

Çok Çekirdekli İşlemciler

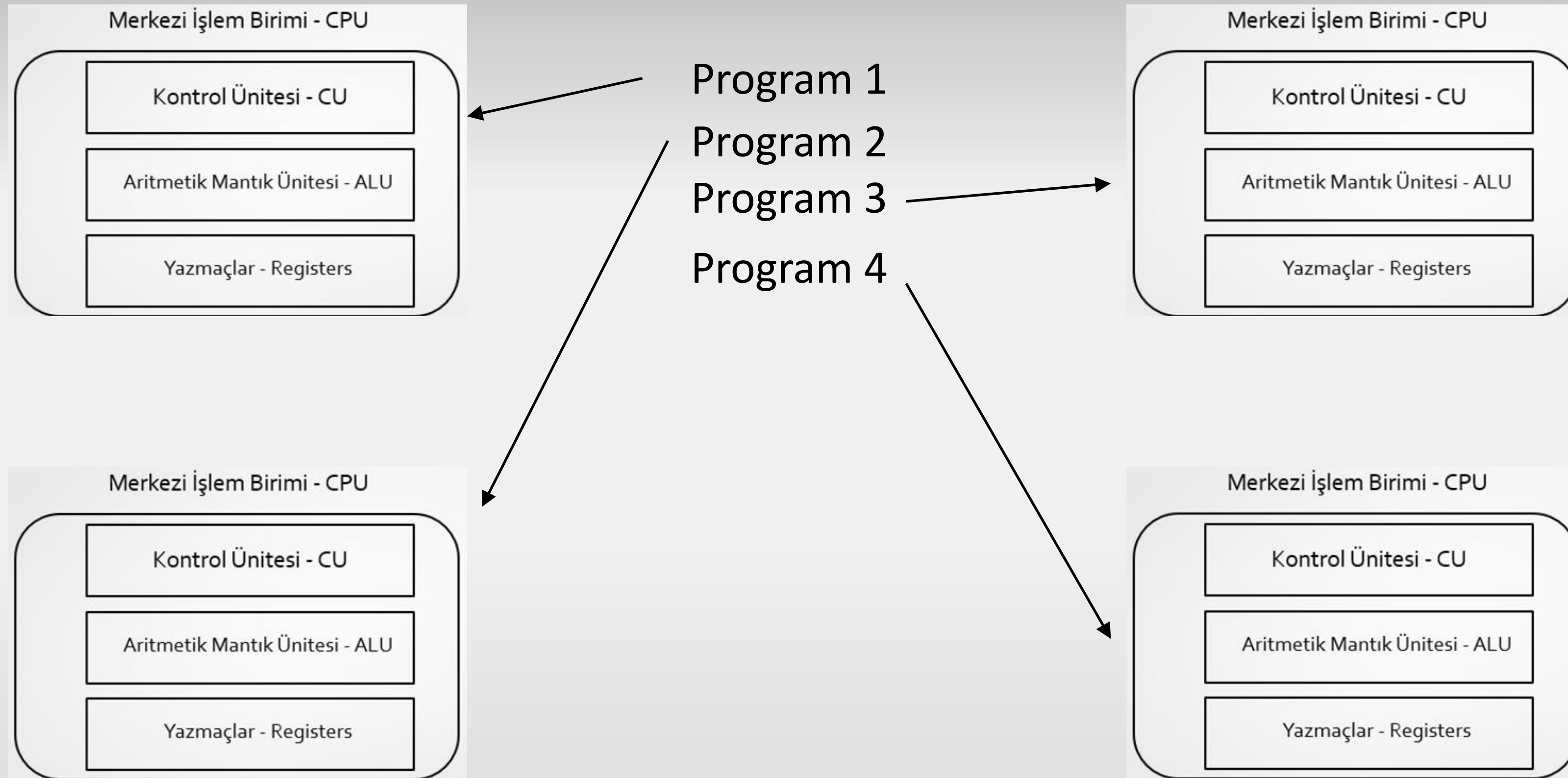
Çok Çekirdekli İşlemciler



Çok Çekirdekli İşlemciler



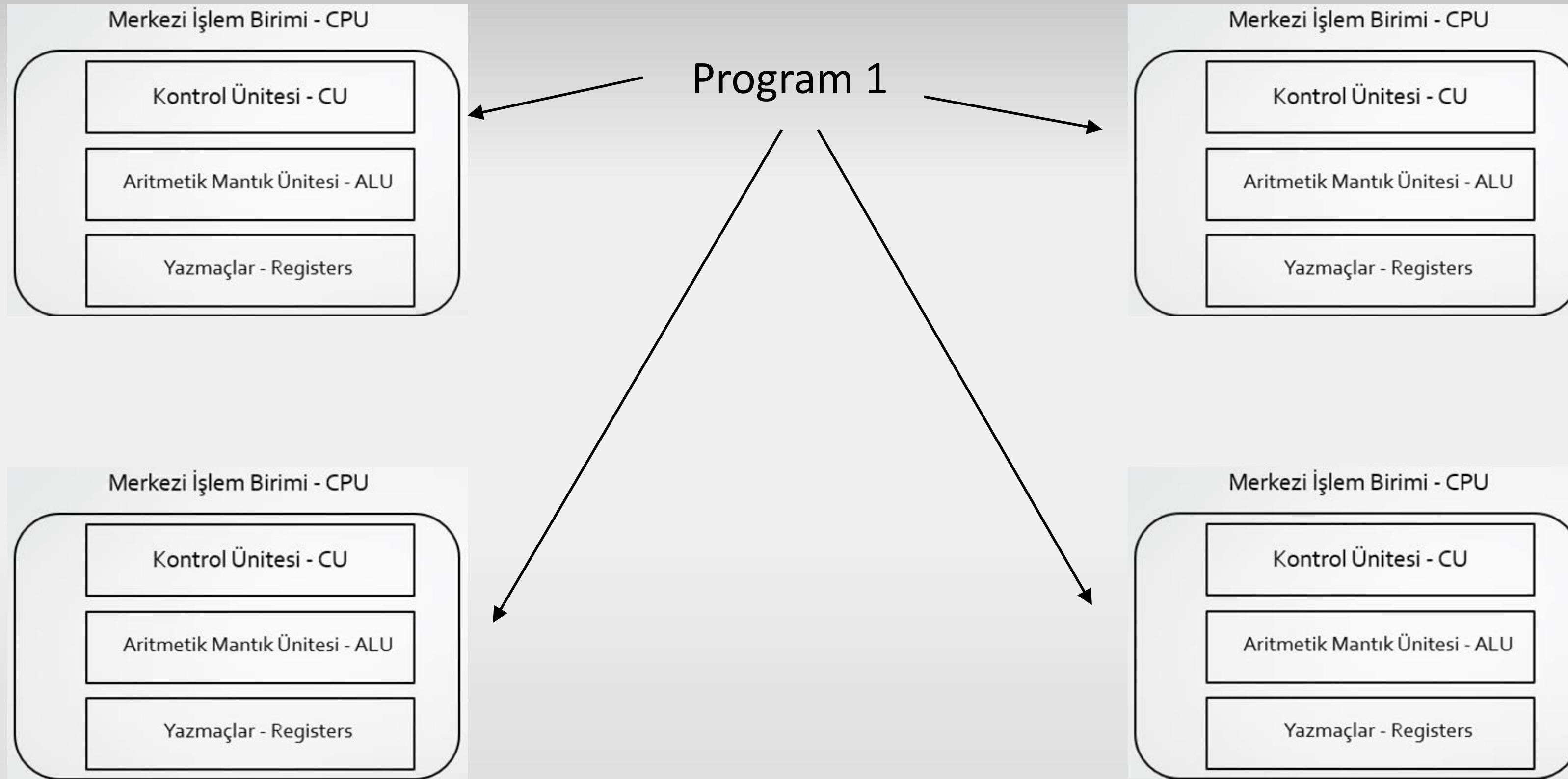
Çok Çekirdekli İşlemciler



Çok Çekirdekli İşlemciler

Program 1

Çok Çekirdekli İşlemciler



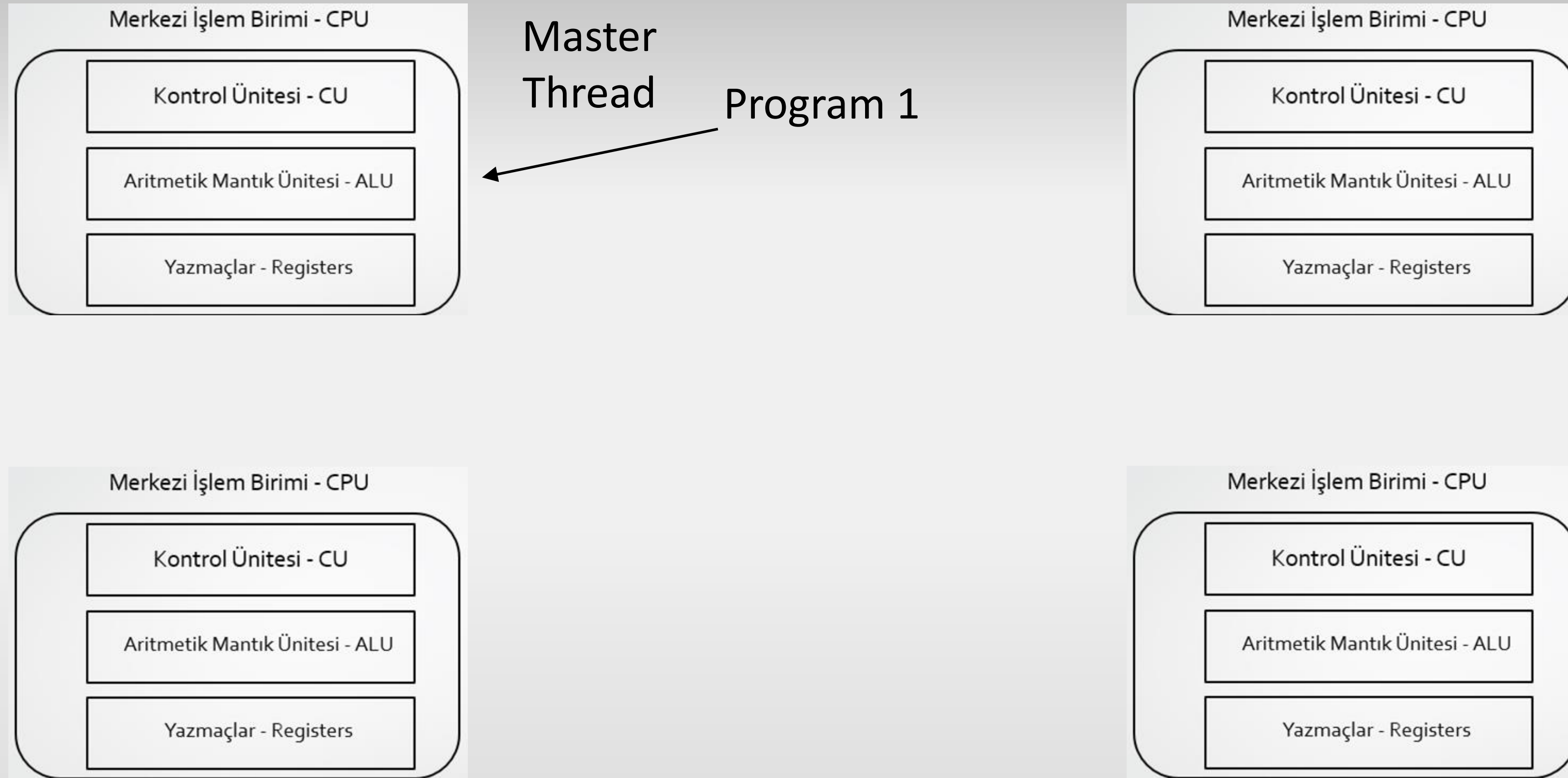
OpenMP

- Fortran, C ve C++ programlama dillerinde kullanılan bir uygulama geliştirme arayüzüdür (API).
- Paralel programlamada kullanılan OpenMP ile bir uygulamadaki komutları birden çok iş parçacığına (thread) paylaştırarak bu iş parçacıklarının eş zamanlı çalışması hedeflenmektedir.
- İş parçacıkları çekirdekleri kullanarak bu paylaşılan komutları çalıştırmaktadır.
- Varsayılan olarak bir uygulama bir ana iş parçacığına (master thread) sahiptir.

CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

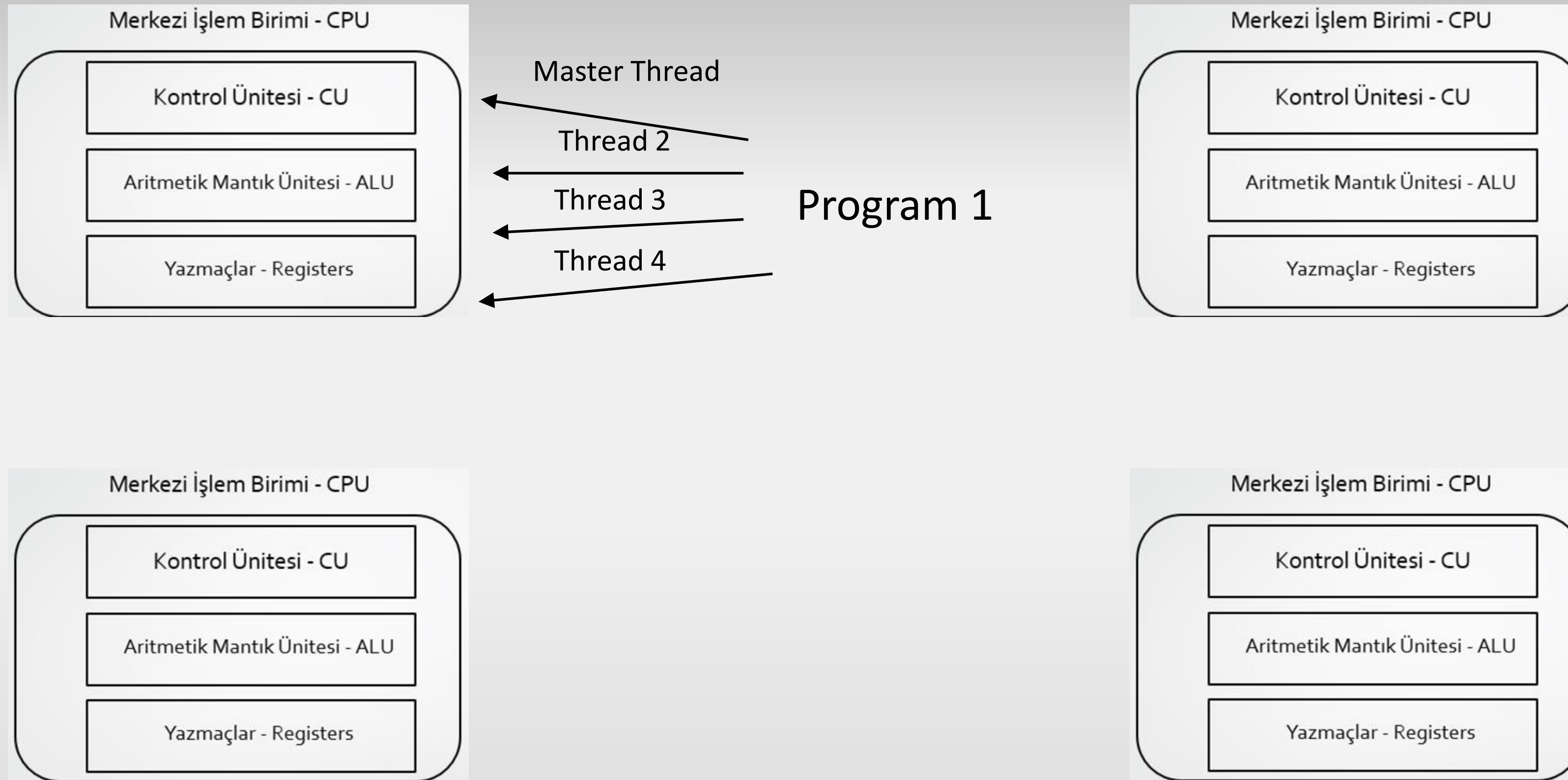
OpenMP



CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

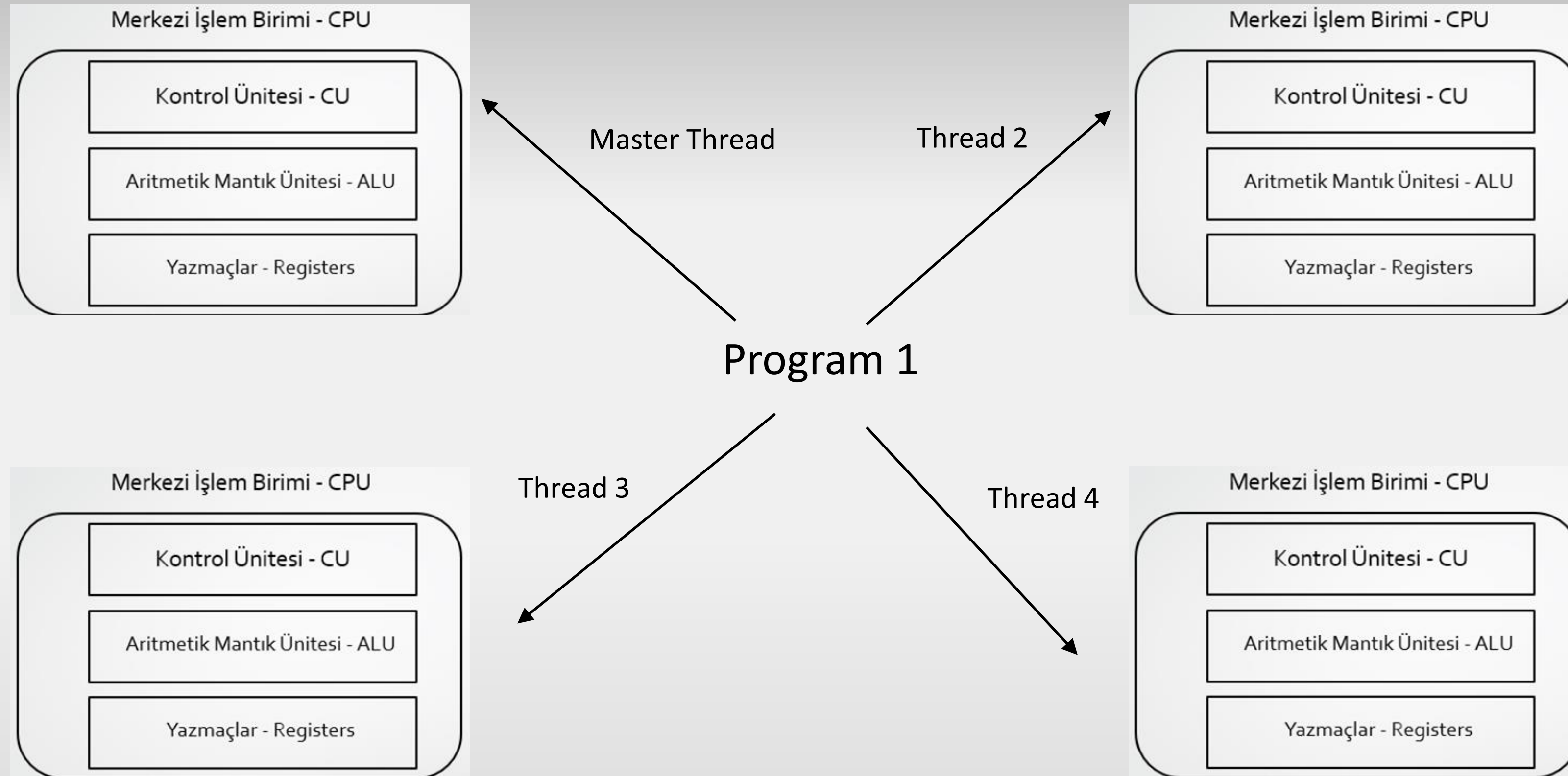
OpenMP



CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

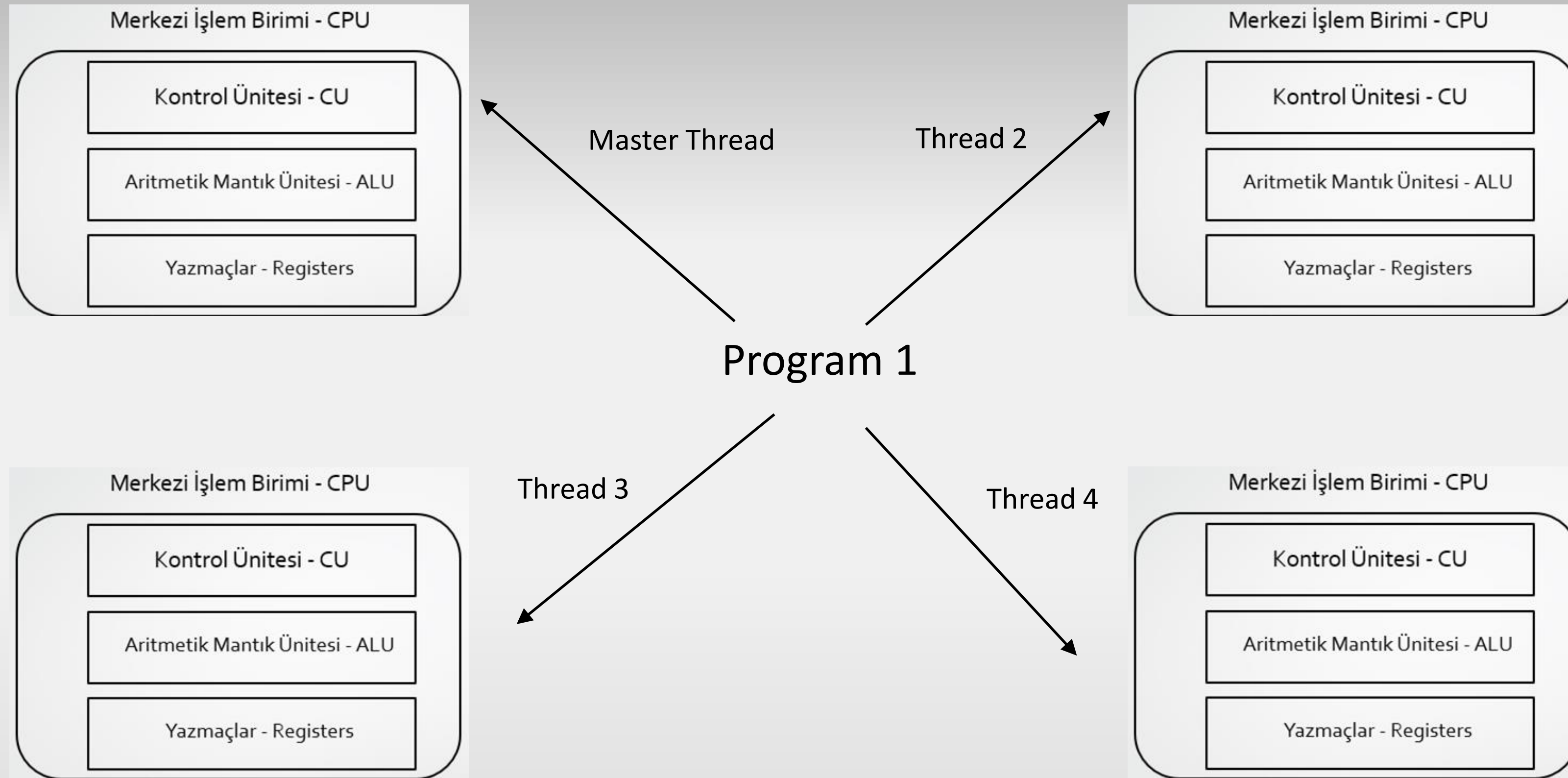
OpenMP



CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

OpenMP



EURO

CUDA Programlama

- C, C++ ve Fortran programlama dilleri ile kullanılan ve grafik işleme biriminde (GPU) fonksiyonların paralel olarak çalışmasını sağlayan bir platformdur
- NVIDIA tarafından geliştirilmiş ve NVIDIA GPU'larında çalışmaktadır
- SIMT (Single Instruction Multiple Thread) modeli şeklinde çalışmaktadır

GPU Mimarisi (Tesla K40)

EURO

PCI Express 3.0 Host Interface

GigaThread Engine

SMX | SMX | SMX | SMX | SMX | SMX | SMX | SMX

L2 Cache

SMX | SMX | SMX | SMX | SMX | SMX | SMX

Memory Controller

Kartın bağlantı noktası

Streaming Multiprocessor (SMX) (15 Tane)

L2 önbelleği

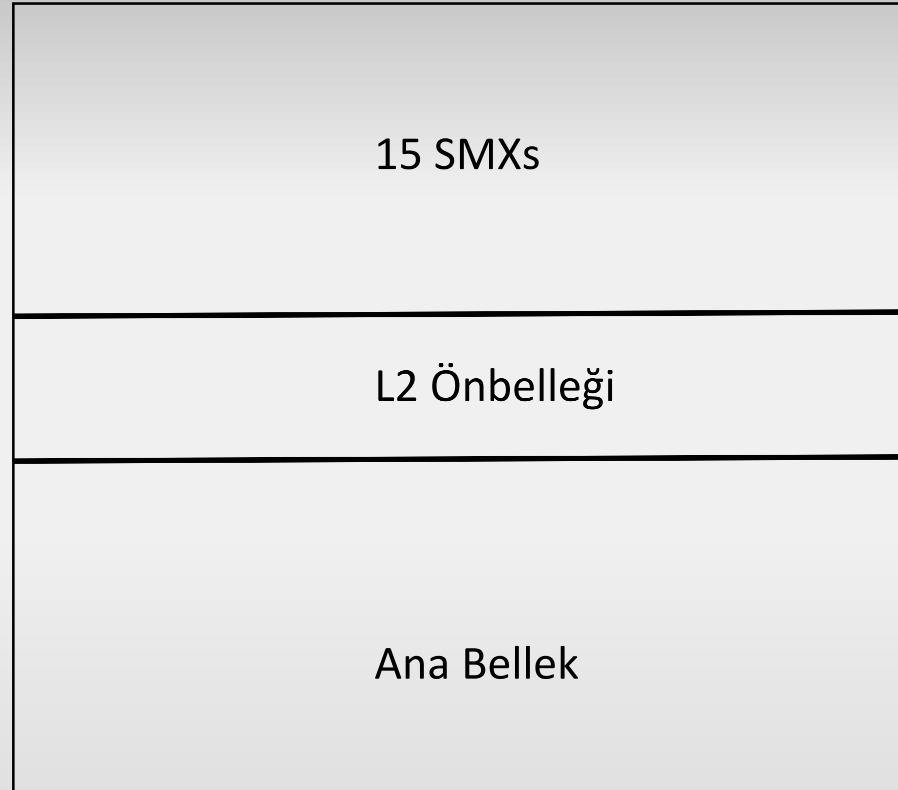
Belleğe erişim noktası

GPU Mimarisi (Tesla K40)

SMX
Instruction Cache
Warp Scheduler | Warp Scheduler | Warp Scheduler | Warp Scheduler
Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch
Register File (65,536 x 32-bit)
Core | DP Unit | LD/ST | SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

- 4 Warp Scheduler
- 32-bit Registers
- 192 Cores (SP floating point units)
- 64 DP floating point units
- 32 Special functional units
- 32 Load/Store units
- 64 KB Shared Memory/L1 cache

GPU Mimarisi (Tesla K40)

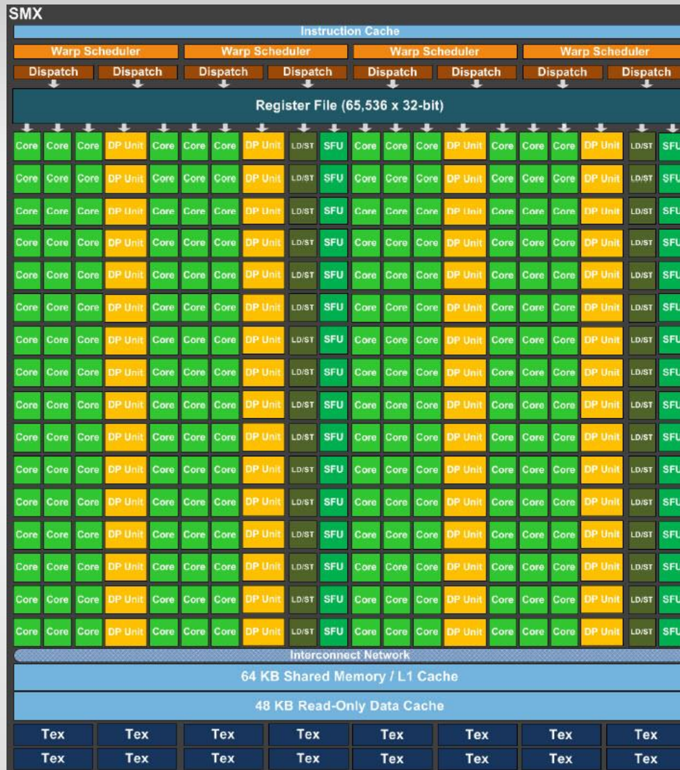


Warp

- Donanımsal bir kavram
- 32 tane iş parçacığından (thread) oluşmaktadır
- Thread'ler yerine warp'lar çekirdeklere çalışmak üzere gönderilir
- Warp içindeki thread'ler birbirleri ile fiziksel olarak bağlıdır
  - Yani bir warp'ın o anki komutu çalıştırmayı bitirmesi için sahip olduğu tüm thread'lerin o komutu çalıştırmayı bitirmesi lazım
  - x = x + 5 + y + sqrtf(x)
    - 3 bellek işlemi, 3 aritmetik işlem ve 1 özel fonksiyonel işlem

Warp

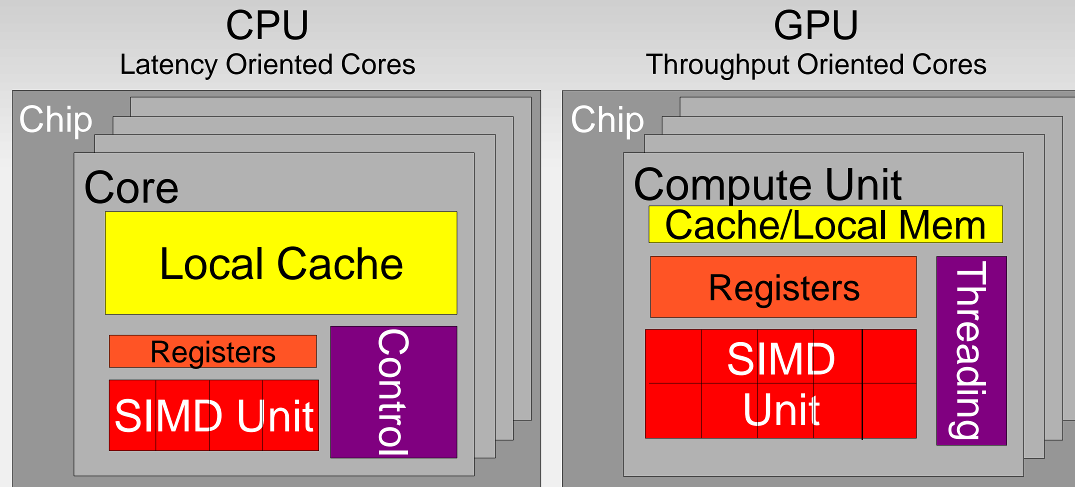
SMX 1



Warp 0 – Komut 1
Warp 1 – Komut 1
Warp 2 – Komut1
Warp 3 – Komut1
Warp 1 – Komut 2
Warp 0 – Komut 2
Warp 2 – Komut 2
Warp 3 – Komut 2
*
*
*
*
*

En fazla 6 warp aynı anda çekirdeklerde çalıştırılabilir

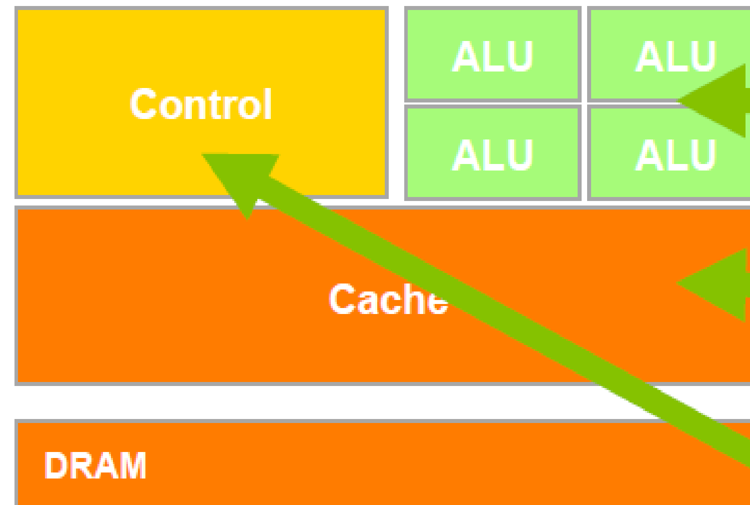
Latency vs Throughput



Örnek:
5 kişi kapasiteli 300 km hızla giden otomobil (Latency oriented)
40 kişi kapasiteli 100 km hızla giden otobüs (Throughput oriented)

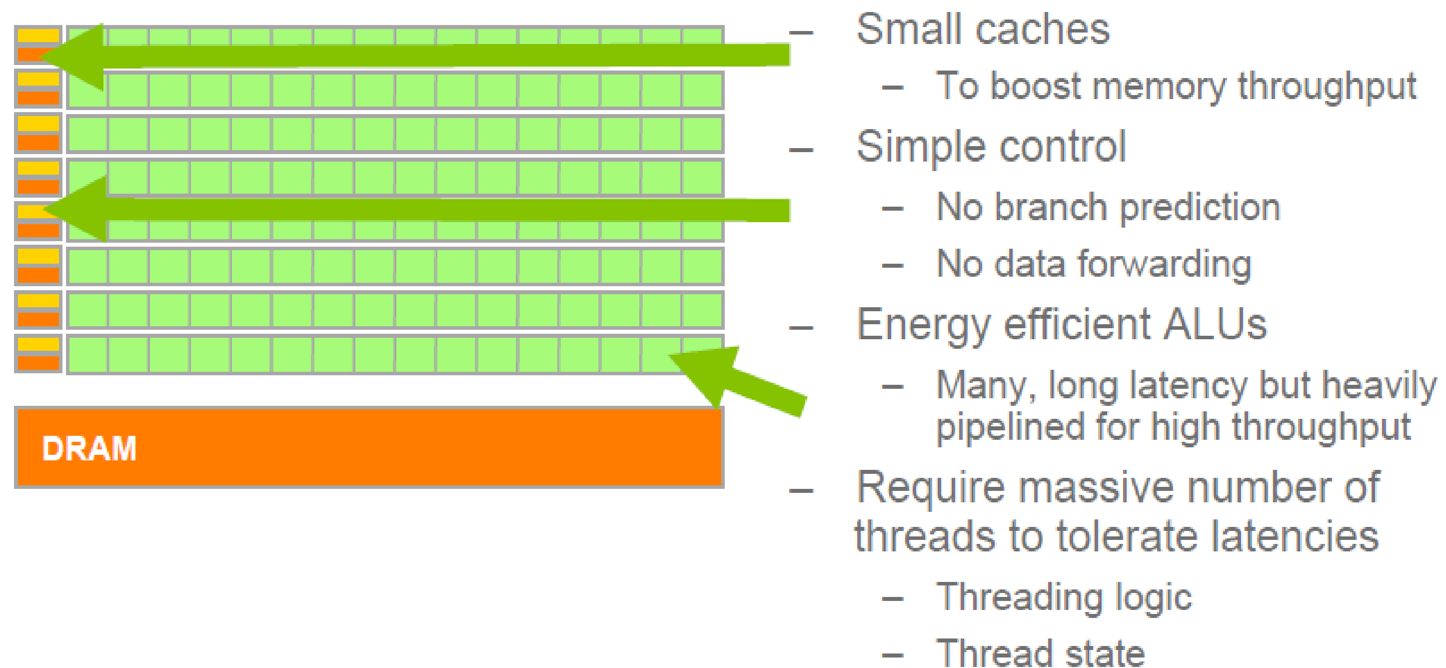
Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)



Ref: GPU Teaching Kit - Accelerated Computing

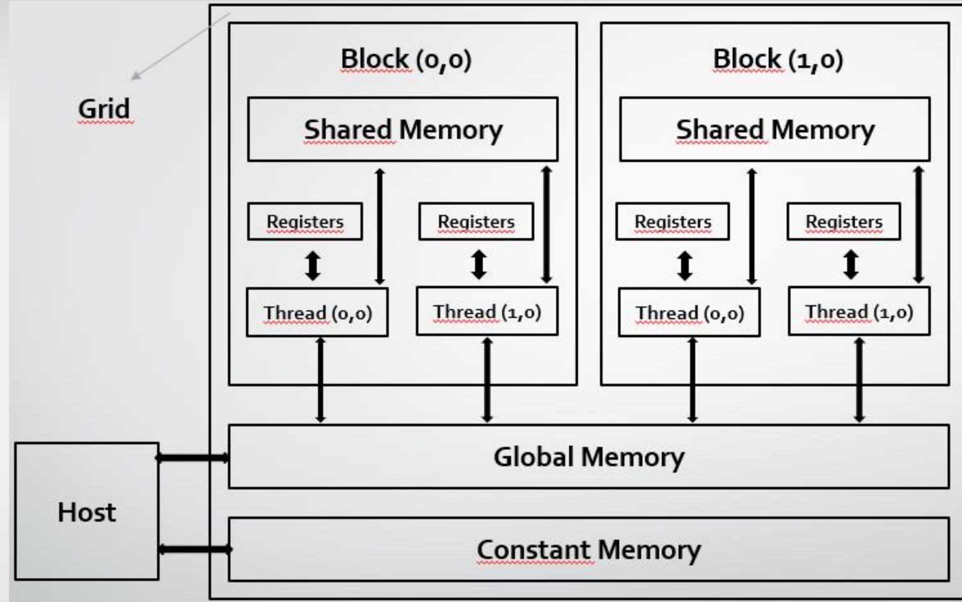
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

CUDA Programlama

- Çok büyük bir veri setinde her bir veri için aynı komut yada komutların çalıştırılması gerektiğinde etkili olan CUDA programlama dili ile kod yazarken etkili bir uygulama için aşağıdaki durumlardan kaçınmalıyız:
  - İç içe çok 'if-else' yapısı olması
  - Komutlar arasındaki bağımlılığın çok olması
  - İç içe döngülerin (for, while) olması
  - Bütün thread'ler arasında senkronizasyonun çok olması
  - Belleğe erişimlerin çok düzensiz olması
  - Her bir thread'in çok farklı türde komutları işlemesi

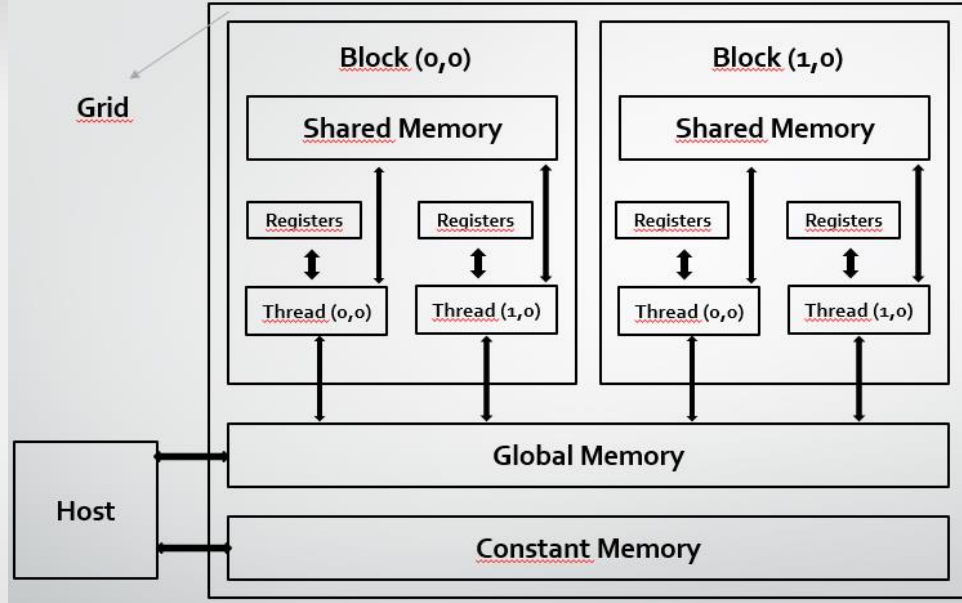
CUDA Programlama Kavramları

CUDA Programlama Kavramları



- Bir grid içinde bloklar oluşturulmaktadır.
- Thread’ler bloklar halinde yaratılmaktadır.
- Her bir thread’in kendine özgü yazmacı (register) ve thread no’su vardır.
- Blok içindeki thread’lerin ortak erişebildiği ortak bellek vardır.
- Aynı blok içindeki thread’ler birbirleri ile senkronize olabilirken, farklı bloklardaki thread’ler birbirleri ile senkronize olamamaktadır.
- Farklı bloklardaki thread’ler aynı L2 önbelleğine ve ana belleğe erişmektedirler.

CUDA Programlama Kavramları



- Bloklar SM'lere eşit şekilde dağılmaktadır.
- Thread'ler warp'lar halinde çekirdeklerde çalıştırılmaktadır.
- Warp donanımsal bir kavram olup kod yazarak oluşturulamamaktadır. Fakat kod içinde dolaylı şekilde kontrol edilebilir.
- "CUDA Kernel" GPU'da çalışan bir fonksiyondur. Oluşturulan bloklar aynı kernel'i çalıştırmaktadır. Tüm bloklar bu kernel'i çalıştırmayı bitirdiğinde fonksiyonun GPU'da çalıştırılması tamamlanmış olmaktadır.

CUDA Kernel

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 using namespace std;
5
6 void my_function()//CPU'da çalışan fonksiyon
7 {
8     printf("Hello World - CPU\n");
9 }
10
11 __global__ void my_kernel()//GPU_da çalışan kernel
12 {
13     printf("Hello World - GPU\n");
14 }
15
16 int main()
17 {
18     my_function();//CPU fonksiyonu çağırılıyor
19
20     my_kernel<<<1,1>>>();//GPU kerneli çalıştırılıyor
21     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
22
23 }
```

__global__ keyword kullanılıyor

Çıktı:
Hello World - CPU
Hello World - GPU

1 blok ve her blokta 1 thread yaratılıyor

Compile: nvcc example1.cu -o example1

CUDA Kernel

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Hello World - GPU - value = %d\n",var_d);
7 }
8
9 int main()
0 {
1     int var = 5;
2     my_kernel<<<1,1>>>(var);//GPU kerneli çalıştırılıyor
3     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
4
5 }
```

GPU’da thread’in yazmacında (register) saklanıyor

Çıktı:
Hello World – GPU – value = 5

CPU belleğinin yığın (stack) kısmında saklanıyor

CUDA Kernel

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Hello World - GPU - value = %d\n",var_d);
7 }
8
9 int main()
10 {
11     int var = 5;
12     my_kernel<<<2,32>>>(var);//GPU kerneli çalıştırılıyor
13     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
14
15 }
```

Çıktı:
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
*
*
*

2 blok yaratılıyor ve her blokta 32 thread yer alıyor. Toplamda 64 thread oluşturuluyor.

CUDA Kernel

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Hello World - GPU - value = %d\n",var_d);
7 }
8
9 int main()
10 {
11     int var = 5;
12     dim3 no_of_blocks(2),no_of_threads(32);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
14     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
15
16 }
```

Çıktı:
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
*
*
*

"dim3" türü değişkenler ile de blok ve thread sayıları tanımlanabilir

CUDA Kernel

```
dim3 no_of_blocks1D(2),no_of_threads1D(32);//1D Boyut
my_kernel<<<no_of_blocks1D,no_of_threads1D>>>(var);//GPU kerneli çalıştırılıyor

dim3 no_of_blocks2D(2,2),no_of_threads2D(4,4);//2D Boyut
my_kernel<<<no_of_blocks2D,no_of_threads2D>>>(var);//GPU kerneli çalıştırılıyor

dim3 no_of_blocks3D(2,2,2),no_of_threads3D(2,2,2);//3D Boyut
my_kernel<<<no_of_blocks3D,no_of_threads3D>>>(var);//GPU kerneli çalıştırılıyor
```

- Blok ve thread sayıları 1D, 2D yada 3D olarak tanımlanabilir
- Yukarıda örnekte üç farklı boyut kullanılarak thread’ler oluşturulmuştur
  - Toplam thread sayıları aynı olmakta
  - Fakat blok sayıları farklıdır
- Sonraki örneklerde boyutlar 1D olacaktır

Built-in Değişkenler

- gridDim: grid'in boyutları
  - gridDim.x
  - gridDim.y
  - gridDim.z
- blockDim: bloğun boyutları
  - blockDim.x
  - blockDim.y
  - blockDim.z
- blockIdx: bloğun grid içindeki indeksleri
  - blockIdx.x
  - blockIdx.y
  - blockIdx.z
- threadIdx: thread'in blok içindeki indeksleri
  - threadIdx.x
  - threadIdx.y
  - threadIdx.z

- Boyutlar 1D, 2D yada 3D olabilir
- Bu değişkenlerin türü "dim3"
- Sadece kernel içinde erişilebilirler

Built-in Değişkenler

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
7 }
8
9 int main()
10 {
11     int var = 5;
12     dim3 no_of_blocks(2),no_of_threads(4);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
14     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
15 }
```

Output:
Block No = 1 - Thread No: 0
Block No = 1 - Thread No: 1
Block No = 1 - Thread No: 2
Block No = 1 - Thread No: 3
Block No = 0 - Thread No: 0
Block No = 0 - Thread No: 1
Block No = 0 - Thread No: 2
Block No = 0 - Thread No: 3

Özcan Dülger, NCC Türkiye

EURO

Built-in Değişkenler

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
7 }
8
9 int main()
10 {
11     int var = 5;
12     dim3 no_of_blocks(2),no_of_threads(4);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
14     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
15 }
```

?

Output:
Block No = 1 - Thread No: 4
Block No = 1 - Thread No: 5
Block No = 1 - Thread No: 6
Block No = 1 - Thread No: 7
Block No = 0 - Thread No: 0
Block No = 0 - Thread No: 1
Block No = 0 - Thread No: 2
Block No = 0 - Thread No: 3

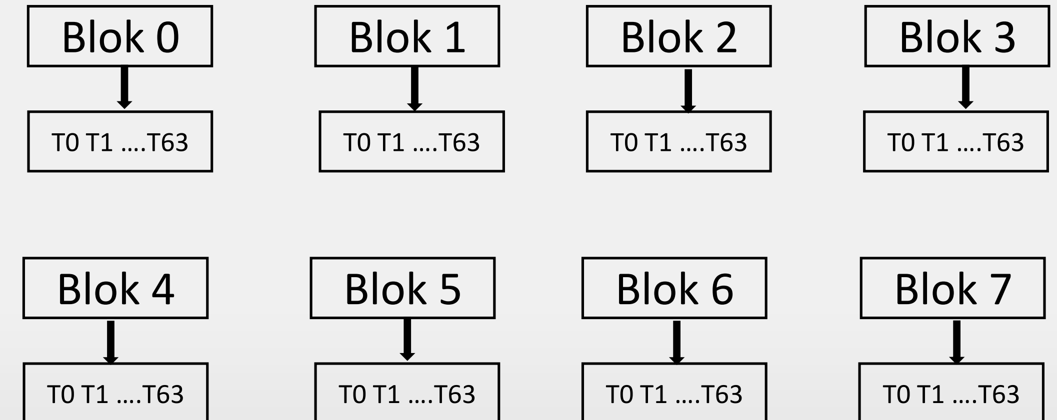
?

- Global thread id'sini nasıl elde ederiz?

Built-in Değişkenler

- Toplam blok = 8
- Blok büyüklüğü = 64
- Toplam thread = 512

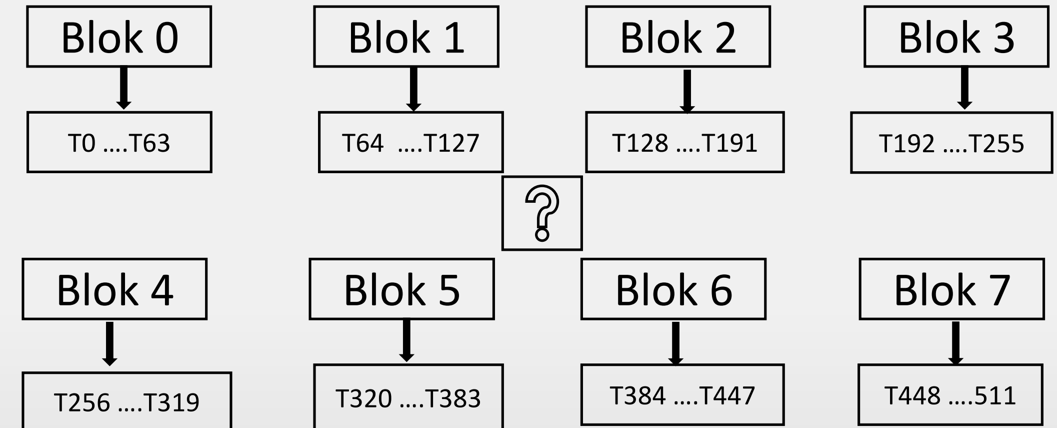
```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
7 }
8
9 int main()
10 {
11     int var = 5;
12     int gridSize = 8,blockSize = 64;
13     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
14     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
15     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
16 }
```



Built-in Değişkenler

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
7 }
8
9 int main()
10 {
11     int var = 5;
12     int gridSize = 8,blockSize = 64;
13     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
14     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
15     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
16 }
```

- Toplam blok = 8
- Blok büyüklüğü = 64
- Toplam thread = 512



CUDA Programlamaya Giriş

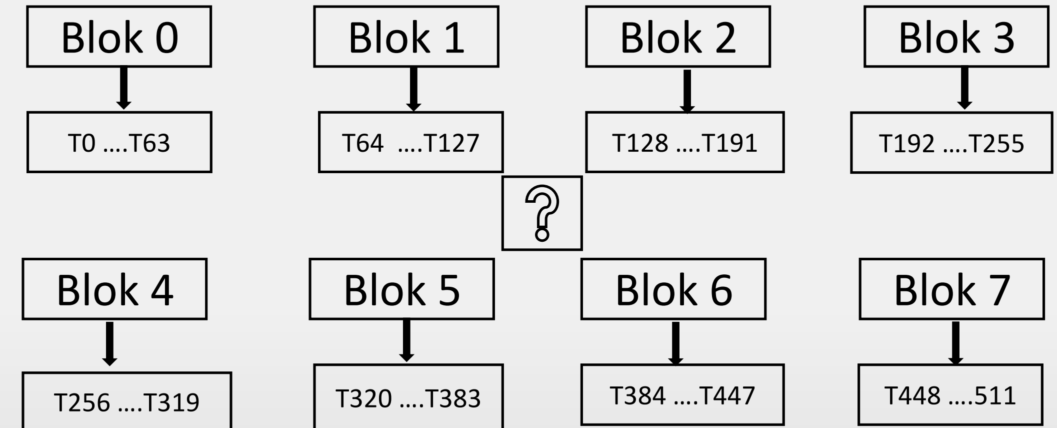
Özcan Dülger, NCC Türkiye

EURO

Built-in Değişkenler

- Toplam blok = 8
- Blok büyüklüğü = 64
- Toplam thread = 512

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
7 }
8
9 int main()
10 {
11     int var = 5;
12     int gridSize = 8,blockSize = 64;
13     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
14     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
15     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
16 }
```



- tid = blockDim.x*blockIdx.x+threadIdx.x;

Built-in Değişkenler

```
1 #include <stdio.h>
2 #include <curand.h>
3
4 __global__ void my_kernel(int var_d)//GPU_da çalışan kernel
5 {
6     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;
7     printf("Block No = %d - Thread No: %d - Global Thread No: %d\n",blockIdx.x,threadIdx.x,tid);
8 }
9
10 int main()
11 {
12     int var = 5;
13     dim3 no_of_blocks(2),no_of_threads(4);
14     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kerneli çalıştırılıyor
15     cudaDeviceSynchronize();//Kernel'in tamamlanması bekleniyor
16 }
```

Output:

Block No = 1 - Thread No: 0 - Global Thread No: 4
Block No = 1 - Thread No: 1 - Global Thread No: 5
Block No = 1 - Thread No: 2 - Global Thread No: 6
Block No = 1 - Thread No: 3 - Global Thread No: 7
Block No = 0 - Thread No: 0 - Global Thread No: 0
Block No = 0 - Thread No: 1 - Global Thread No: 1
Block No = 0 - Thread No: 2 - Global Thread No: 2
Block No = 0 - Thread No: 3 - Global Thread No: 3

CUDA Bellek İşlemleri

```
 1 #include <stdio.h>
 2
 3 void my_function(int *A,int size)
 4 {
 5     for(int i=1;i<=size;i++)
 6         printf("A[i] = %d\n",A[i-1]);
 7 }
 8
 9 int main()
10 {
11     int size = 1000;//Dizi büyüklüğü
12     int *A_Host;
13     A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
14
15     for(int i=1;i<=size;i++)//Diziye başlangıç değerleri atanıyor
16         A_Host[i-1] = i;
17
18     my_function(A_Host,size);//Fonksiyon çağırılıyor
19
20     delete[] A_Host;//Dizi bellekten siliniyor
21 }
```

CUDA Bellek İşlemleri

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int *A,int size)//CUDA kernel
4 {
5     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
6     printf("A[tid] = %d\n",A[tid]);
7 }
8
9 int main()
10 {
11     int size = 1000;//Dizi büyüklüğü
12     int ThreadPerBlock = 64;//Blok büyüklüğü. 32'nin katı olması iyi olur
13     int BlockPerGrid = (size-1)/ThreadPerBlock+1;//Blok sayısı
14     int *A_Host;
15     A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
16
17     for(int i=1;i<=size;i++)//Diziye başlangıç değerleri atanıyor
18         A_Host[i-1] = i;
19
20     int *A_GPU;
21     cudaMalloc(&A_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor
22     cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
23
24     dim3 DimBlock(ThreadPerBlock);//Bir bloktaki thread sayısı
25     dim3 DimGrid(BlockPerGrid);//Bir griddeki blok sayısı
26
27     my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel çalıştırılıyor
28     cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı
29
30     delete[] A_Host;//Dizi CPU belleğinden siliniyor
31     cudaFree(A_GPU);//Dizi GPU belleğinden siliniyor
32 }
```

- cudaMalloc(void** address, size_t size)
  - 1.girdi ana bellekte ayrılacak alanın adresi
  - 2.girdi ayrılacak alanın bayt cinsinden büyüklüğü
- cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
  - 1.girdi verilerin kopyalanacağı alanın adresi
  - 2.girdi kopyalanacak verilerin adresi
  - 3.girdi kopyalancak verinin bayt cinsinden büyüklüğü
  - 4.girdi veri kopyalamanın yönü
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice
- cudaFree(void* address)
  - GPU'daki anabellekteki ilgili ana temizler

CUDA Bellek İşlemleri

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int *A,int size)//CUDA kernel
4 {
5     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
6     printf("A[tid] = %d\n",A[tid]);
7 }
8
9 int main()
10 {
11     int size = 1000;//Dizi büyüklüğü
12     int ThreadPerBlock = 64;//Blok büyüklüğü. 32'nin katı olması iyi olur
13     int BlockPerGrid = (size-1)/ThreadPerBlock+1;//Blok sayısı
14     int *A_Host;
15     A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
16
17     for(int i=1;i<=size;i++)//Diziye başlangıç değerleri atanıyor
18         A_Host[i-1] = i;
19
20     int *A_GPU;
21     cudaMalloc(&A_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor
22     cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
23
24     dim3 DimBlock(ThreadPerBlock);//Bir bloktaki thread sayısı
25     dim3 DimGrid(BlockPerGrid);//Bir griddeki blok sayısı
26
27     my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel çalıştırılıyor
28     cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı
29
30     delete[] A_Host;//Dizi CPU belleğinden siliniyor
31     cudaFree(A_GPU);//Dizi GPU belleğinden siliniyor
32 }
```

- Bir warp'ta thread sayısının 32 olduğunu ve warp'ların çekirdeklere çalışmak üzere gönderildiğini düşününce kaynakları verimli kullanmak için bir bloktaki thread sayısını 32'nin katları şeklinde tanımlamamız gerekmektedir.

- 15 blok yaratırsak dizinin her bir elemanı için bir thread yaratamayacağımızdan dolayı 16 blok yaratmamız gerekmektedir.
- Veri boyutu blok büyüklüğünün katı olmadığı için özel bir formulasyona ihtiyacımız vardır.
  - (size-1)/ThreadPerBlock+1

CUDA Programlamaya Giriş

Özcan Dülger, NCC Türkiye

CUDA Bellek İşlemleri

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int *A,int size)//CUDA kernel
4 {
5     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
6     if(tid<size)
7         printf("A[tid] = %d\n",A[tid]);
8 }
9
10 int main()
11 {
12     int size = 1000;//Dizi büyüklüğü
13     int ThreadPerBlock = 64;//Blok büyüklüğü. 32'nin katı olması iyi olur
14     int BlockPerGrid = (size-1)/ThreadPerBlock+1;//Blok sayısı
15     int *A_Host;
16     A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
17
18     for(int i=1;i<=size;i++)//Diziye başlangıç değerleri atanıyor
19         A_Host[i-1] = i;
20
21     int *A_GPU;
22     cudaMalloc(&A_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor
23     cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
24
25     dim3 DimBlock(ThreadPerBlock);//Bir bloktaki thread sayısı
26     dim3 DimGrid(BlockPerGrid);//Bir griddeki blok sayısı
27
28     my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel çalıştırılıyor
29     cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı
30
31     delete[] A_Host;//Dizi CPU belleğinden siliniyor
32     cudaFree(A_GPU);//Dizi GPU belleğinden siliniyor
33 }
```

- 16 blok yaratıldığında 1024 tane thread çalışıyor olmaktadır. Dizinin büyüklüğü 1000 olduğu için 24 tane thread okuma işlemi yaparken “out of bound” hatasına yakalanacaktır.
- “If” koşulu eklenerek bu durumun önünce geçilebilir

EURO

CUDA Değişken Türleri

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Not: Thread'in oluşturduğu diziler ana bellekte (global memory) oluşturulmaktadır.

CUDA Bellek İşlemleri

```
1 #include <stdio.h>
2
3 const int ThreadPerBlock = 64;//Blok büyüklüğü. 32'nin katı olması iyi olur
4
5 __global__ void my_kernel(int *A,int size)//CUDA kernel
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8
9     __shared__ int shared_A[ThreadPerBlock];//Shared dizi tanımlandı. Boyutu bloktaki thread sayısına eşit
10    shared_A[threadIdx.x] = A[tid];//Her thread ana bellekteki değerini ortak belleğe kopyalıyor
11
12    if(tid<size)
13        printf("A[tid] = %d\n",shared_A[threadIdx.x]);//Ortak bellek üzerinden işlem yapılıyor
14 }
15
16 int main()
17 {
18    int size = 1000;//Dizi büyüklüğü
19    int BlockPerGrid = (size-1)/ThreadPerBlock+1;//Blok sayısı
20    int *A_Host;
21    A_Host = new int[size];//CPU belleğinde (Heap bölgesi) yer açılıyor
22
23    for(int i=1;i<=size;i++)//Diziye başlangıç değerleri atanıyor
24        A_Host[i-1] = i;
25
26    int *A_GPU;
27    cudaMalloc(&A_GPU,sizeof(int)*size);//GPU ana belleğinde yer açılıyor
28    cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//CPU'dan GPU'ya veri aktarımı
29
30    dim3 DimBlock(ThreadPerBlock);//Bir bloktaki thread sayısı
31    dim3 DimGrid(BlockPerGrid);//Bir griddeki blok sayısı
32
33    my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel çalıştırılıyor
34    cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//GPU'dan CPU'ya veri aktarımı
35
36    delete[] A_Host;//Dizi CPU belleğinden siliniyor
37    cudaFree(A_GPU);//Dizi GPU belleğinden siliniyor
38 }
```

Thread Senkronizasyonu

```
 5 __global__ void my_kernel(int *A,int size)//CUDA kernel
 6 {
 7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
 8 
 9     __shared__ int shared_A[ThreadPerBlock];//Shared dizi tanımlandı. Boyutu bloktaki thread sayısına eşit
10     shared_A[threadIdx.x] = A[tid];//Her thread ana bellekteki değerini ortak belleğe kopyalıyor
11 
12     __syncthreads();//Blok içindeki tüm thread'ler bu noktaya ulaşınca sonraki işlemler başlıyor
13 
14     //shared_A dizisi üzerinde tüm thread'ler ortak işlem yapmakta
15 }
```

Özcan Dülger, NCC Türkiye

CUDA Çalışma Modeli

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

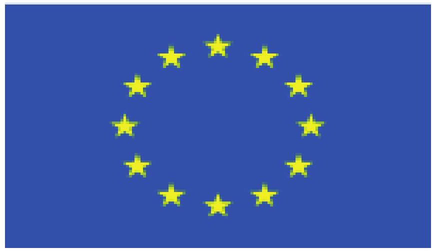
Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

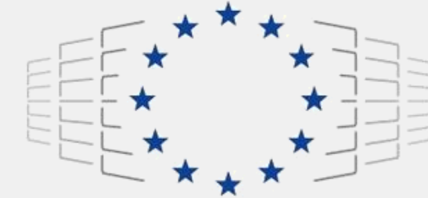
Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

Blocking-Non-Blocking Fonksiyonlar

- Blocking (Senkronize) Fonksiyonlar: GPU’da çalışması bittikten sonra program CPU’ya dönüyor:
  - cudaMalloc
  - cudaMemcpy
  - cudaDeviceSynchronize
  - cudaMallocHost
  - cudaFree
- Non-Blocking (Asenkronize) Fonksiyonlar: GPU’da çalışması bitmeden program CPU’ya dönüyor:
  - CUDA Kernel
  - cudaMallocAsync
  - cudaMemcpyAsync
  - cudaMemset
  - cudaEventRecord

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-1

- Konu: Vector Addition
- 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)