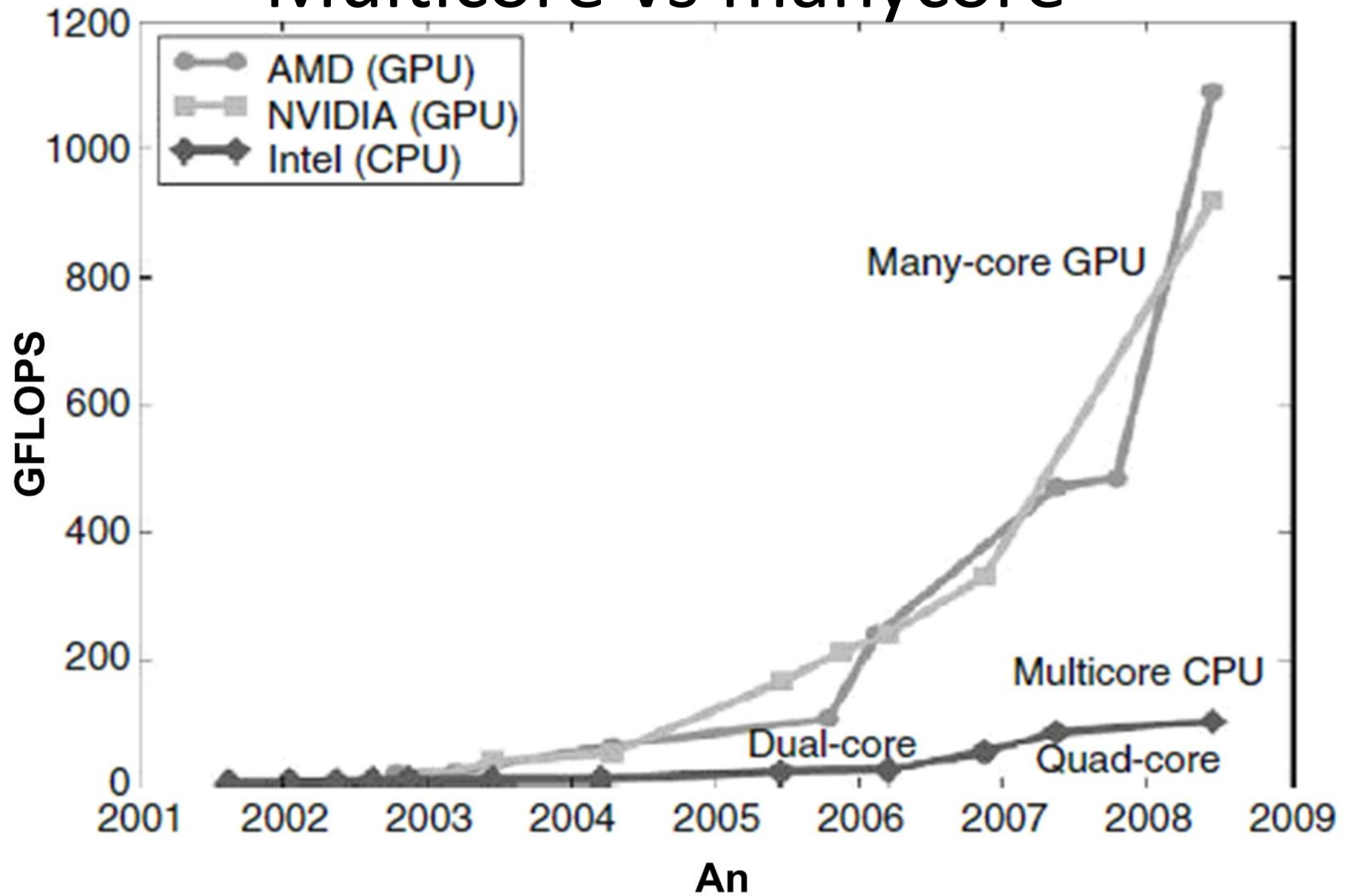


Procesare paralelă utilizând GPU

Introducere în CUDA

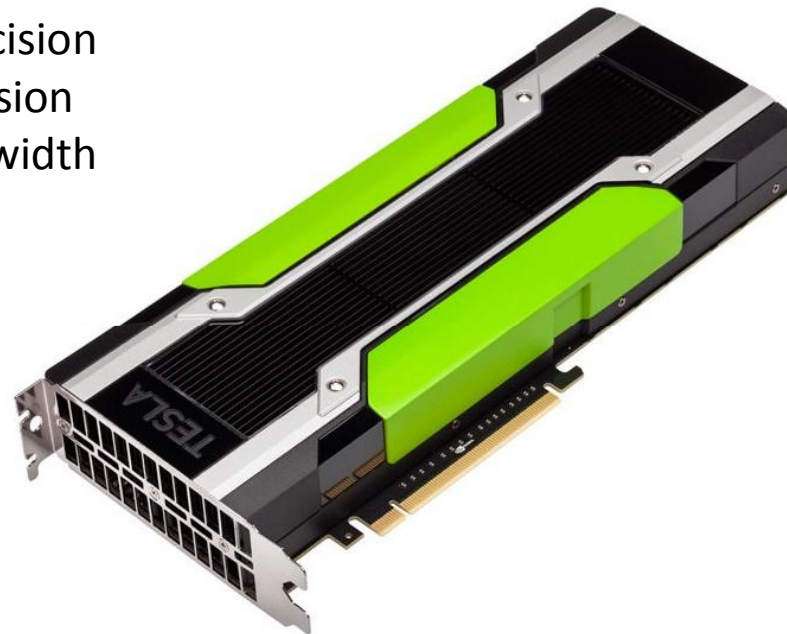
Cosmin – Ioan Niță

Multicore vs manycore

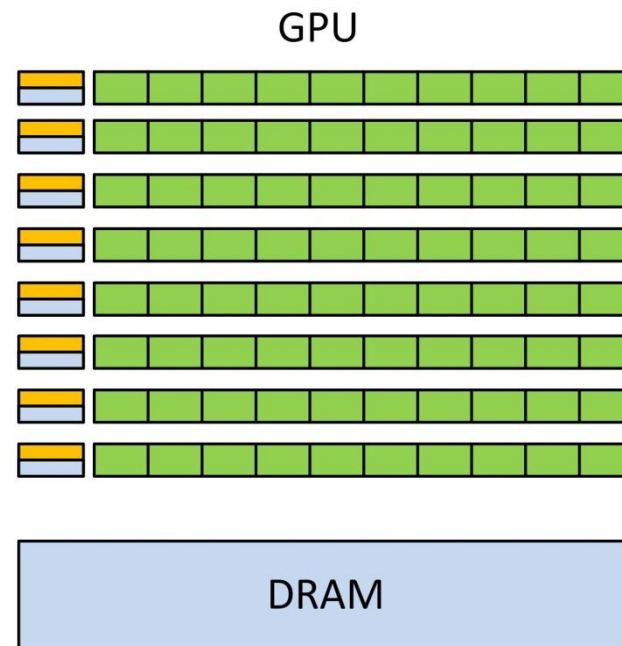
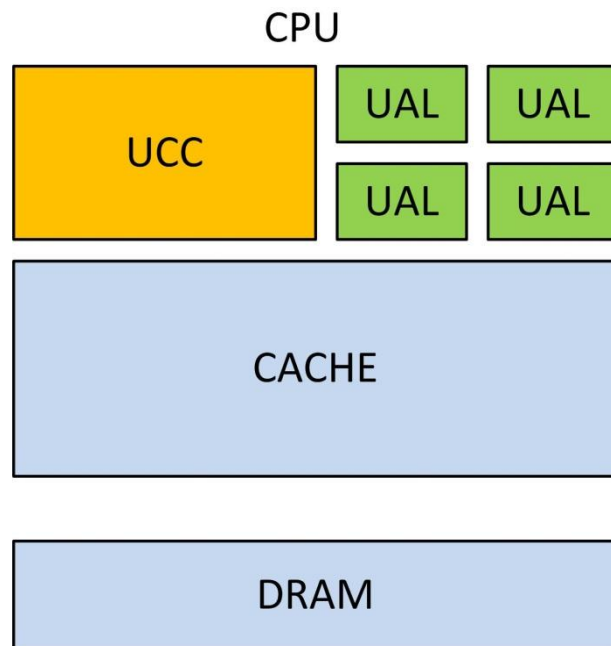


Nvidia TESLA k80

- 2.91 TFLOPS double precision
- 8,74 TFLOPS single precision
- 480 GB/s memory bandwidth
- 24 GB memorie

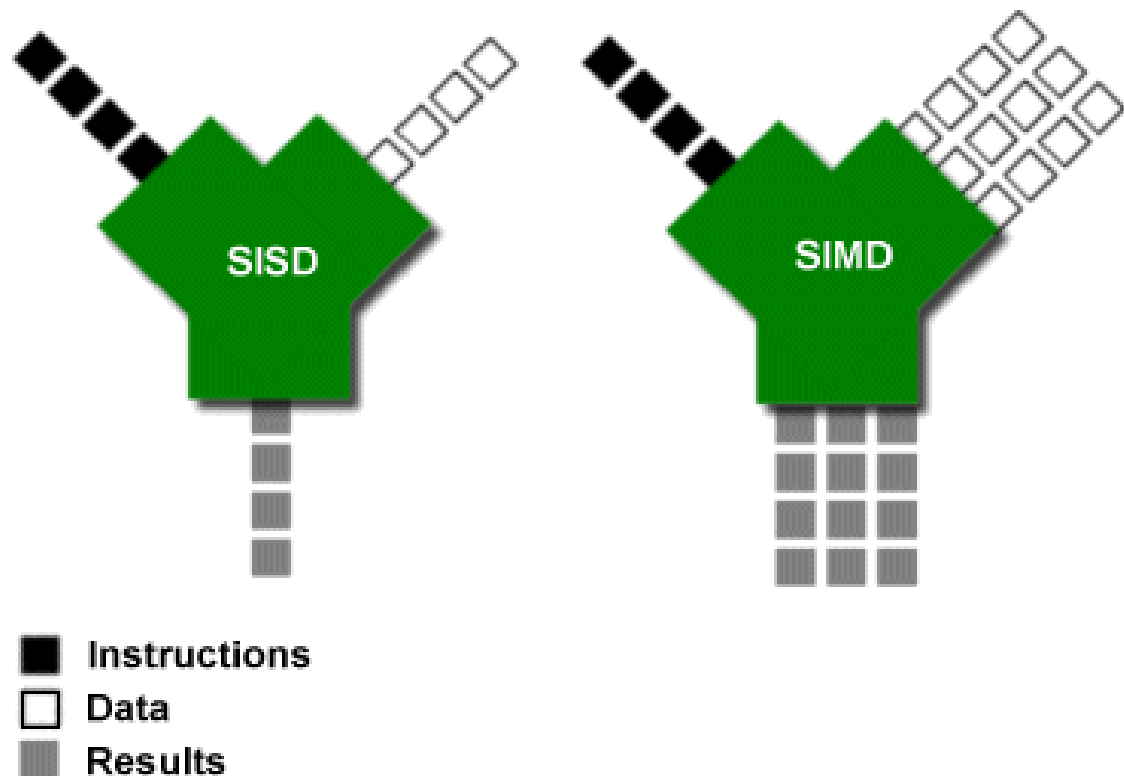


Multicore vs manycore

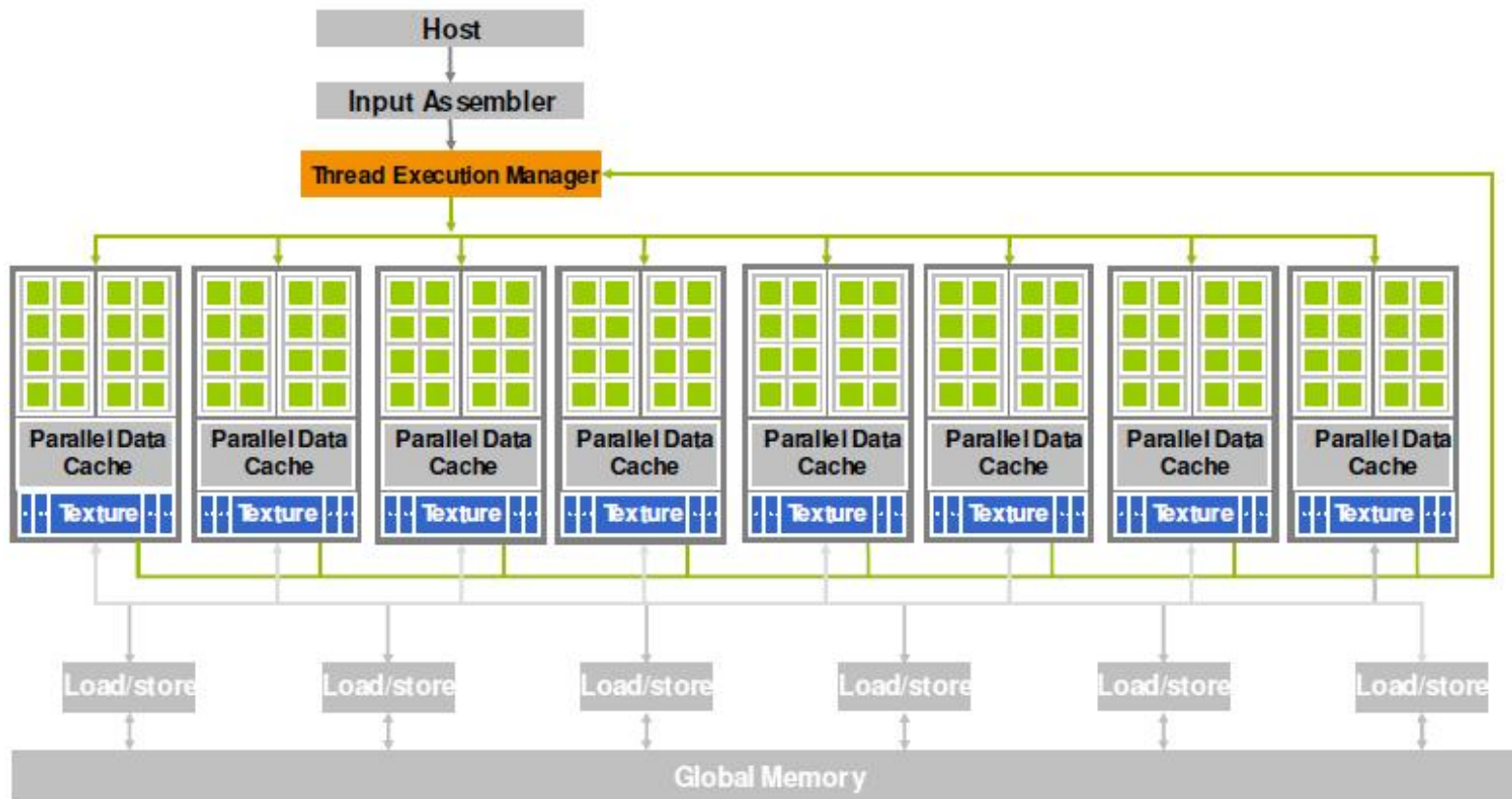


Multicore vs manycore

- *SISD* – Single instruction single data
- *SIMD* – Single instruction multiple data



Arhitectura unui GPU



Limbaajul CUDA

- Expune paralelismul unui GPU pentru calcule de uz general
- Bazat pe limbajul C++, conține doar câteva extensii în plus
- Conține funcții pentru management-ul dispozitivelor CUDA și a memoriei

Terminologie

- Host: CPU și memoria lui (memorie host)
- Device: GPU și memoria lui (device memory)

Host



Device

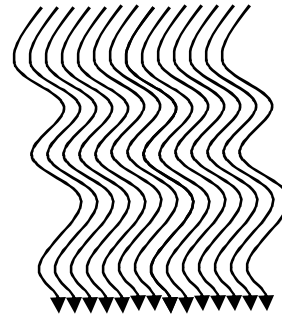


Structura unui program CUDA

Cod secvențial



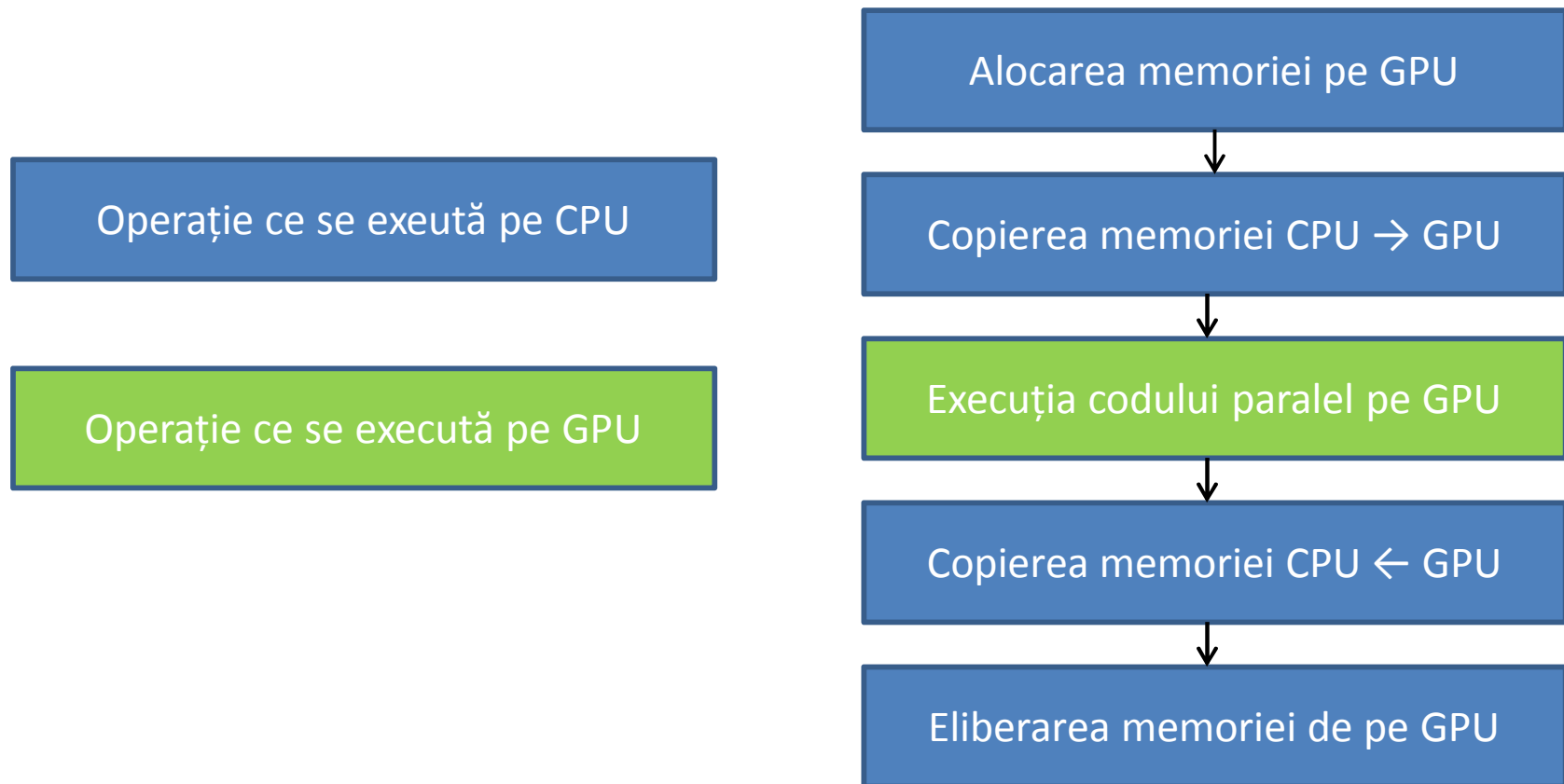
Cod paralel



Cod secvențial



Structura unui program CUDA



Managementul memoriei

- Spațiul de memorie al GPU este diferit de cel al CPU → se alocă memorie separat pentru GPU
- GPU nu poate accesa memoria host
- CPU poate accesa memoria GPU doar prin funcții speciale
- Librăria CUDA conține funcții pentru managementul memoriei de pe GPU:
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar cu echivalentul de pe CPU: `malloc()`, `free()`, `memcpy()`

Alocarea memoriei

- Host:

```
float * floats_h = new float[N];  
float * floats_h = (float*)malloc(N * sizeof(float));
```

- Device:

```
float * floats_d;  
cudaMalloc((void**)&floats_d, N*sizeof(float));
```

Dimensiunea memoriei alocate se dă în octeți

Copierea memoriei

- De la host la device:

```
cudaMemcpy(floats_d, floats_h, N*sizeof(float),  
cudaMemcpyHostToDevice);
```

- De la device la host:

```
cudaMemcpy(floats_h, floats_d, N*sizeof(float),  
cudaMemcpyDeviceToHost);
```

Primul parametru este tot timpul
destinația iar al 2-lea este sursa

Kernel-ul CUDA

- Kernel-ul: funcție ce se este apelată de pe CPU și se execută pe GPU
- Implementarea porțiunii de cod ce se va executa pe GPU se face într-un kernel
- Definirea unui kernel:

```
__global__ void exemplu_kernel(float * p1, float * p2, int p3)
{

}
```

Kernel-ul CUDA

- Lansarea în execuție:

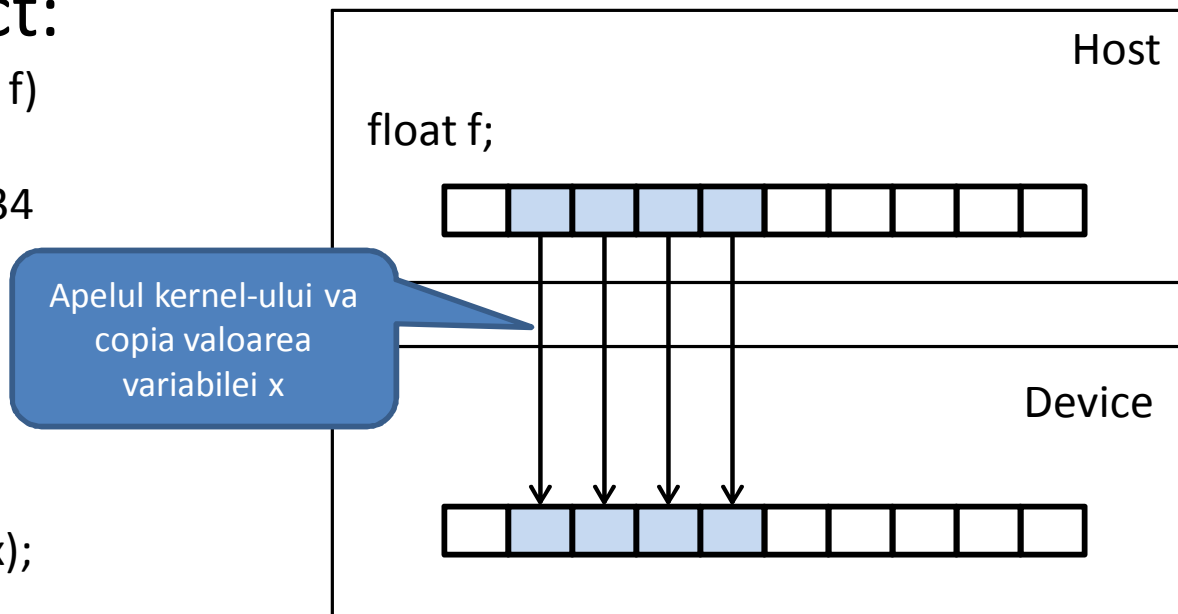
```
exemplu_kernel <<<1, 1 >>>( [lista de parametrii] );
```

Transferul parametrilor la un kernel CUDA

- Într-un kernel CUDA nu se pot folosi decât variabile ce se află în memoria device
- Toți parametrii unui kernel sunt copiați automat în memoria device la momentul lansării în execuție
- Transferul direct:

```
__global__ void ceva(float f)
{
    // Aici f va fi 1.234
}
```

```
void main()
{
    float x = 1.234;
    ceva<<<1,1>>>(x);
}
```



Transferul parametrilor la un kernel CUDA

- Transferul prin pointer:

```
__global__ void ceva(float *f)
```

```
{
```

```
...
```

```
}
```

```
void main()
```

```
{
```

```
float *f_dev;
```

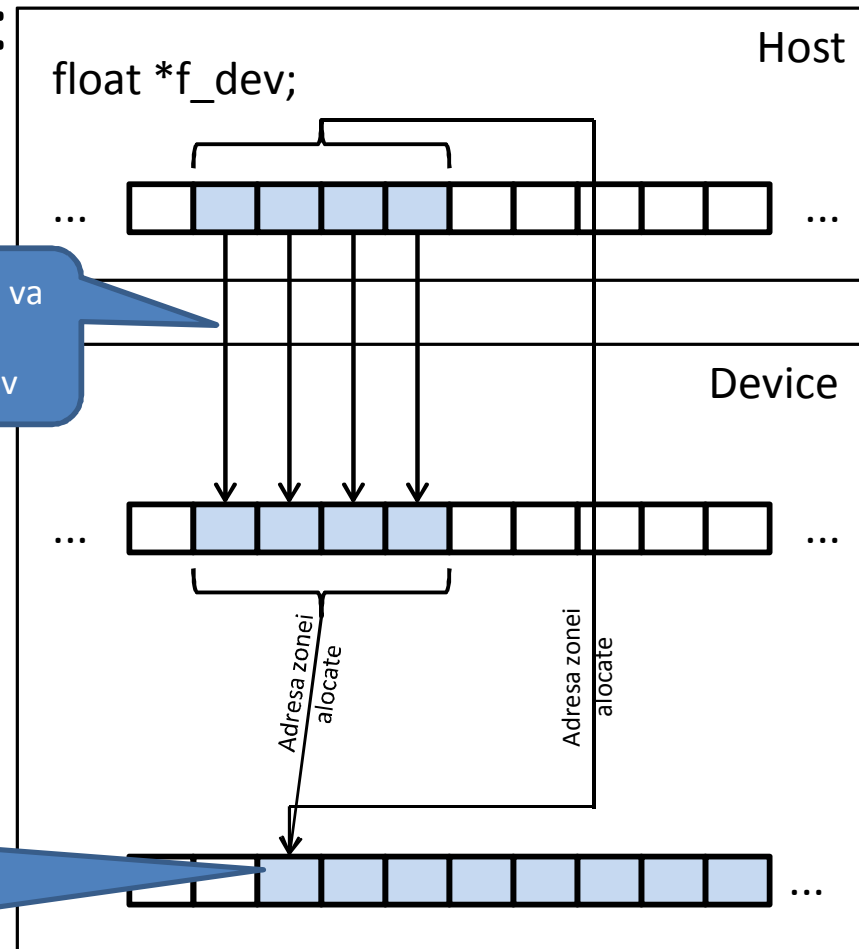
```
cudaMalloc((void**)&f_dev, ...)
```

```
ceva<<<1,1>>>(f_dev);
```

```
}
```

Apelul kernel-ului va copia **valoarea** pointerului `f_dev`

`cudaMalloc` va alocă această zonă de memorie iar adresa primului element va fi atribuită pointerului `f_dev` (de pe host)



Transferul parametrilor la un kernel CUDA

- Transferul prin referință

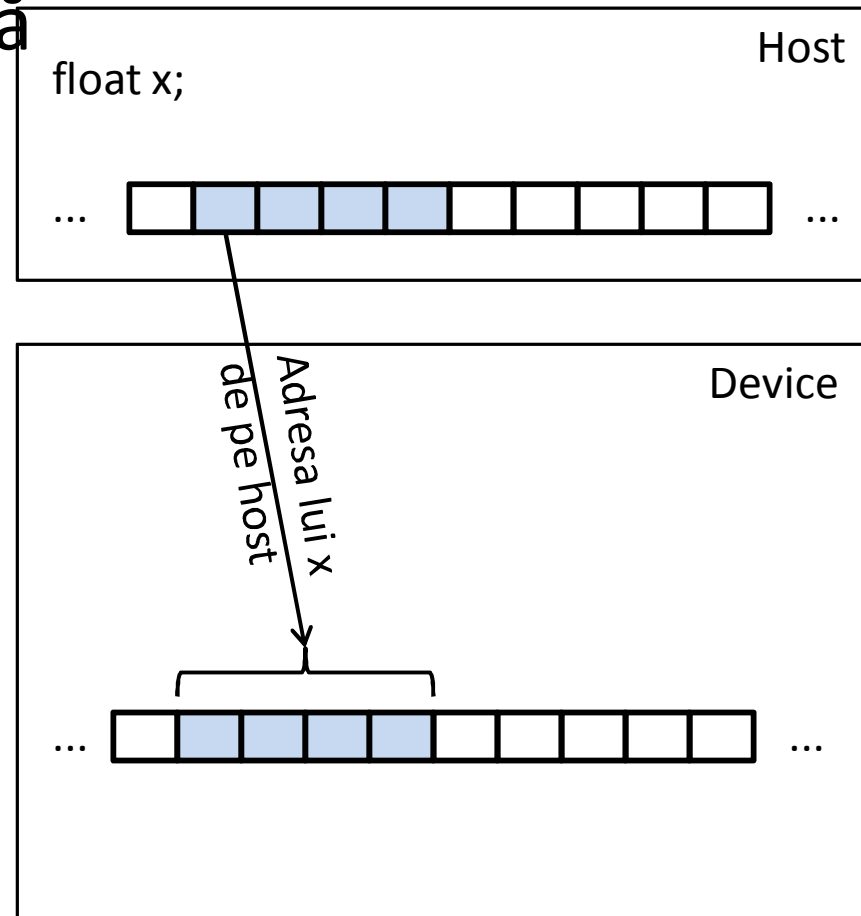
```
__global__ void ceva(float &f)
{
    // Ce valoare are f?
}
```

```
void main()
{
    float x = 1.234;
    ceva<<<1,1>>>(x);
}
```

Transferul parametrilor la un kernel CUDA

- Transferul prin referință

```
__global__ void ceva(float &f)  
{  
    // Ce valoare are f?  
}  
void r  
{  
}  
}
```



Transferul parametrilor la un kernel CUDA

- Transferul instanței unei structuri.

```
struct A
{
    int i;
    float x;
    double y;
}
__global__ void ceva(A a)
{
    ...
}

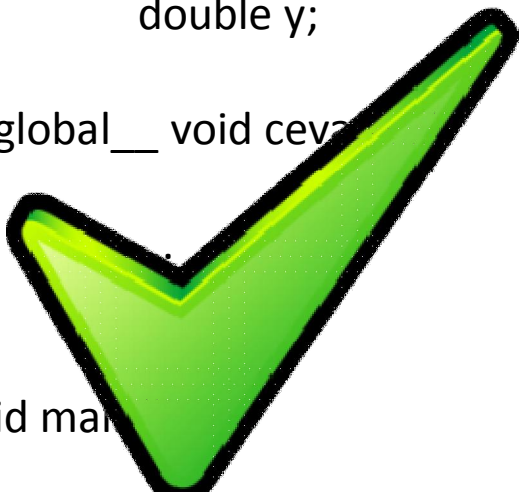
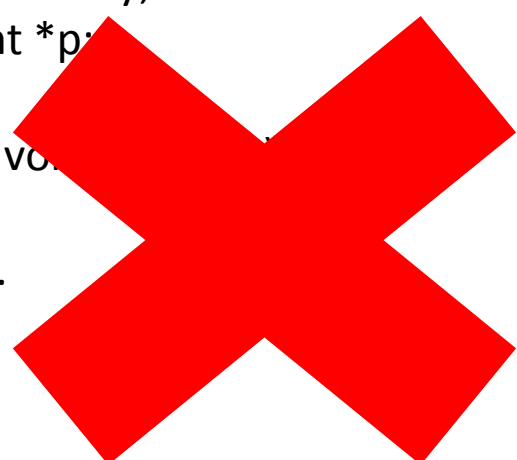
void main()
{
    A a;
    a.i = 0; a.x = 1.0f; a.y = 2.0;
    ceva<<<1,1>>>(a);
}
```

```
struct A
{
    int i;
    float x;
    double y;
    int *p;
}
__global__ void ceva(A a)
{
    ...
}

void main()
{
    A a;
    a.i = 0; a.x = 1.0f; a.y = 2.0; p = new int;
    ceva<<<1,1>>>(a);
}
```

Transferul parametrilor la un kernel CUDA

- Transferul instanței unei clase.

<pre>struct A { int i; float x; double y; } __global__ void ceva { ... } void main() { A a; a.i = 0; a.i = 1.0f; a.y = 2.0; ceva<<<1,1>>>(a); }</pre> 	<pre>struct A { int i; float x; double y; int *p; } __global__ void { ... } void main() { A a; a.i = 0; a.i = 1.0f; a.y = 2.0; p = new int; ceva<<<1,1>>>(a); }</pre> 
--	--

Exemplu

```
__global__ void test(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

```
int main()
{
    int a = 1, b = 2, c;
    int *a_dev, *b_dev, *c_dev;

    //Alocare pe GPU
    cudaMalloc((void**)&a_dev, sizeof(int));
    cudaMalloc((void**)&b_dev, sizeof(int));
    cudaMalloc((void**)&c_dev, sizeof(int));

    //Copiere CPU → GPU
    cudaMemcpy(a_dev, &a, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, &b, sizeof(int), cudaMemcpyHostToDevice);

    //Lansarea in executie a kernel-ului
    test << <1, 1 >> >(a_dev, b_dev, c_dev);

    //Copere GPU → GPU
    cudaMemcpy(&c, c_dev, sizeof(int), cudaMemcpyDeviceToHost);

    printf("a+b = %d\n", c);

    return 0;
}
```

```
int main()
{
    int a = 1, b = 2, c;

    c = a + b;

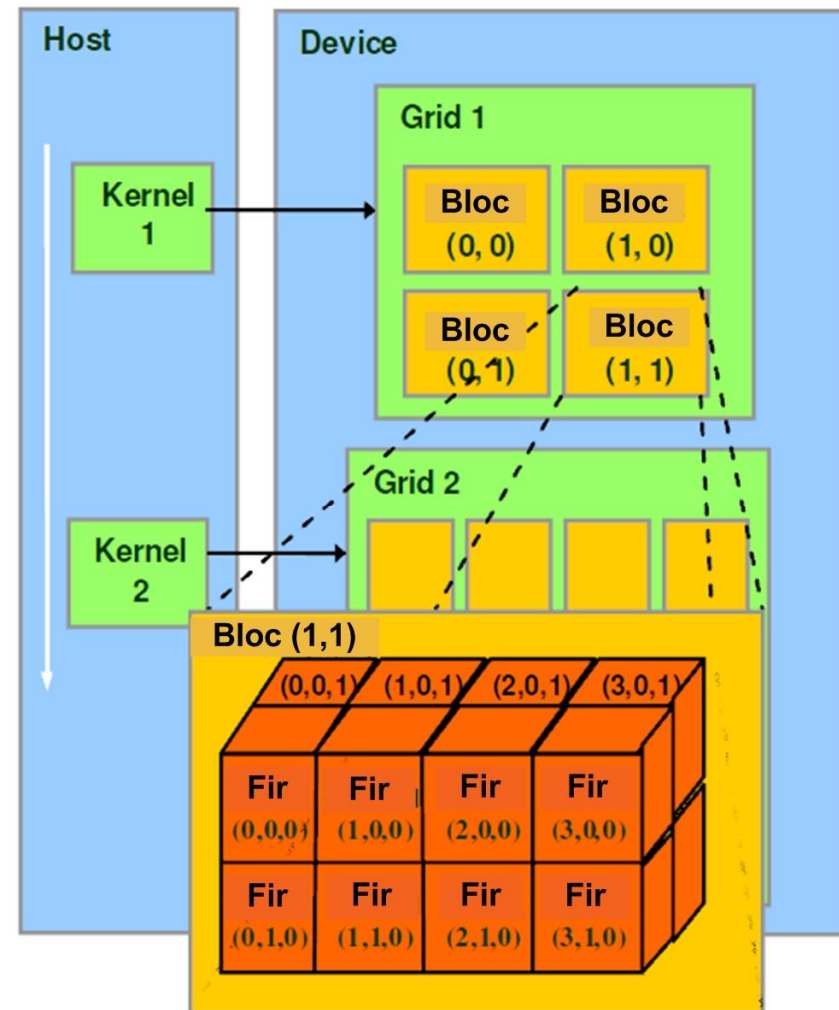
    printf("a+b = %d\n", c);

    return 0;
}
```



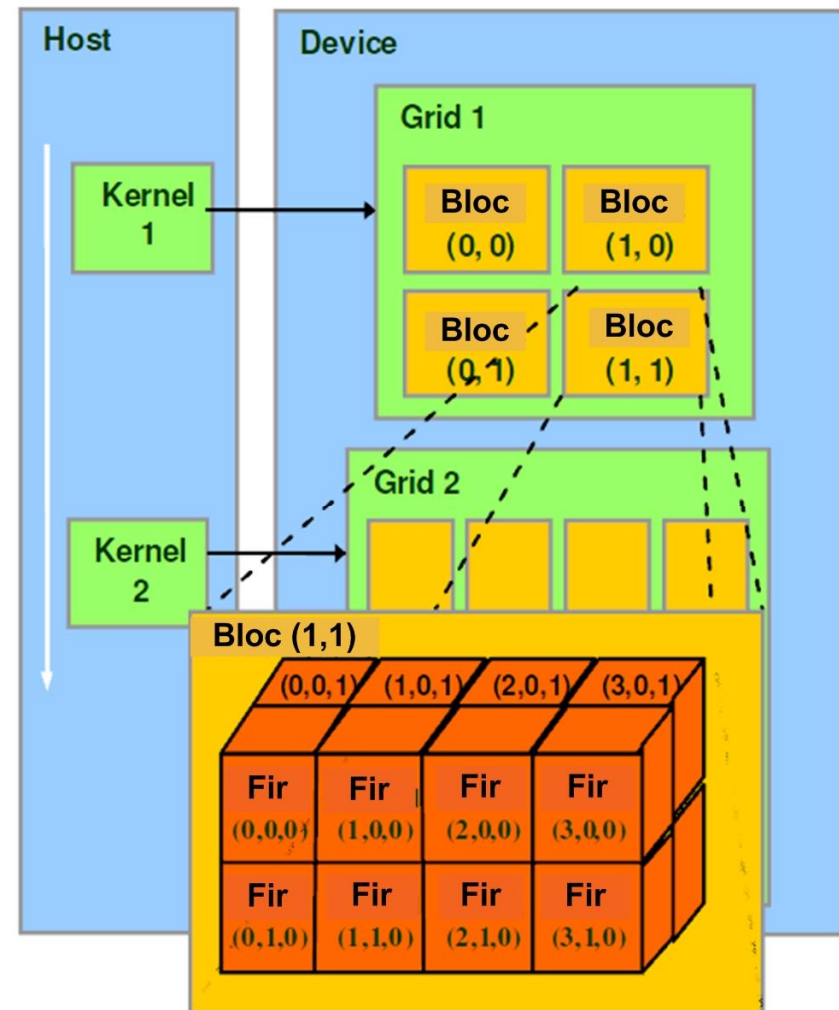
Organizarea firelor de execuție CUDA

- *Block* și *Grid*
- Pot avea una, două sau trei dimensiuni
- Lungimea se specifică la lansarea în execuție



Organizarea firelor de execuție CUDA

- Un bloc are dimensiunile maxime 1024 x 1024 x 64, **până la maxim 1024 fire pe bloc in total**
- Un grid poate avea maxim 65535^3 blocuri



Paralelismul CUDA

- Variabile preefinite în orice funcție kernel
 - **threadIdx**: id-ul thread-ului curent în bloc
 - **blockIdx**: id-ul blocului curent în grid
 - **gridDim**: dimensiunea grid-ului (nr total de blocuri)
 - **blockDim**: dimensiunea unui bloc (nr de fire într-un bloc)
- Aceste variabile sunt de tip *dim3* (au ca membrii x,y și z pentru a descrie o rețea 3D de fire). Exemplu:
threadIdx.x, threadIdx.y, threadIdx.z

```
__global__ void exemplu_kernel()  
{  
    int thread_id = threadIdx.x;  
    int blockid = blockIdx.x;  
}
```

Paralelismul CUDA

- Lansarea în execuție:

```
exemplu kernel <<<lungime_grid, lungime_block >>>( [lista de parametrii] );
```

- Se lansează în execuție *lungime_grid*lungime_bloc* fire de execuție

- D.p.d.v. logic este echivalent cu:

```
for (int i = 0; i < lungime_grid*lungime_block; i++)  
{  
    exemplu_kernel( [lista de parametrii] );  
}
```

Exemplu – Adunarea a doi vectori

Cod C standard

```
void sum_serial(int n,
float * a,
float * b,
float * c
)
{
for (int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}

sum_serial(4096 * 256, a, b, c);
```

CUDA

```
__global__
void sum_parallel(int n,
float * a,
float * b,
float * c
)
{
int i = blockIdx.x*blockDim.x +
threadIdx.x;
c[i] = a[i] + b[i];
}

sum_parallel<<<4096, 256>>>(4096 *
256, a, b, c);
```

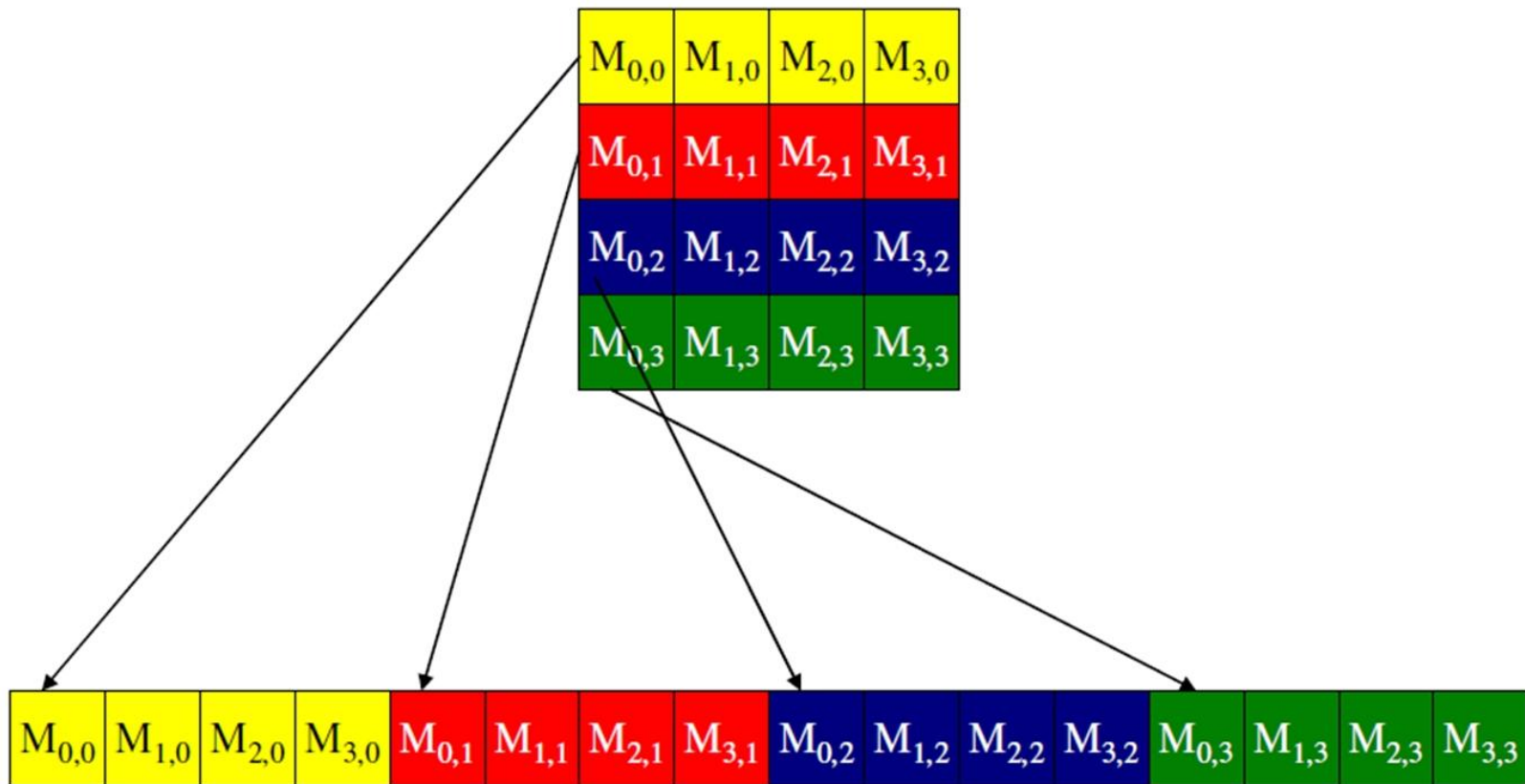
Lucrul cu date multi-dimensionale

Două variante:

- Reducere a dimensionalității datelor.
- Grid de thread-uri multi-dimensional

Lucrul cu date multi-dimensionale

- Exemplu: Înmulțirea matrice-vector



Ordine liniarizată pe baza adreselor crescătoare