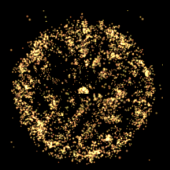


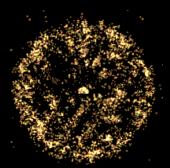
GPU, mint szuperszámítógép – II. (1)



Grafikus kártyák, mint olcsó szuperszámítógépek - II.

tanuló szeminárium

Jurek Zoltán, Tóth Gyula
SZFKI, Röntgendiffrakciós csoport



Vázlat

I.

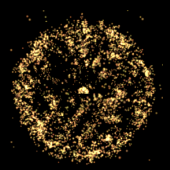
- Motiváció
- Beüzemelés
- C alapok
- CUDA programozási modell, hardware adottságok

II.

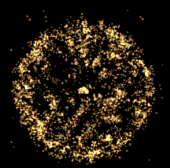
- Tippek (pl. több GPU egyidejű használata)
- Könyvtárak (cuFFT, cuBLAS, ...)
- GPU számolás CUDA programozás nélkül

III.

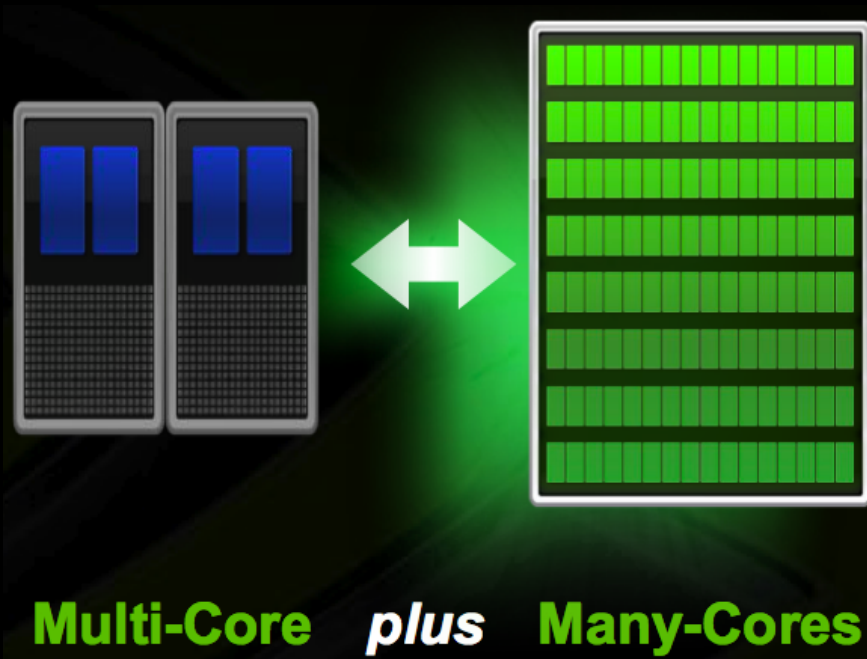
- Optimalizálás
- ...



Emlékeztető



Heterogén számolások



Host computer + Device

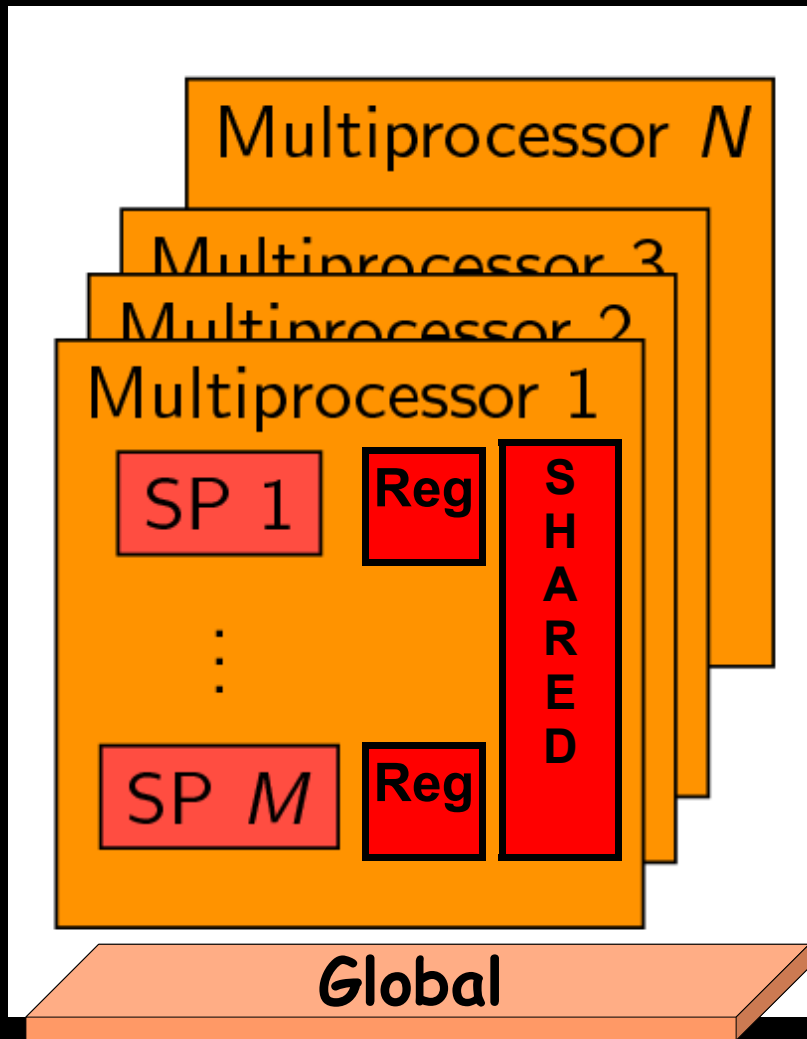
Grafikus kártya

~ coprocesszor:

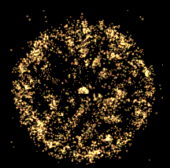
- saját processzorok
- saját memória
- host másol rá adatot
- host indítja a számolást rajta
- host visszamásolja az eredményt
- **CUDA** modell +
C kiterjesztés



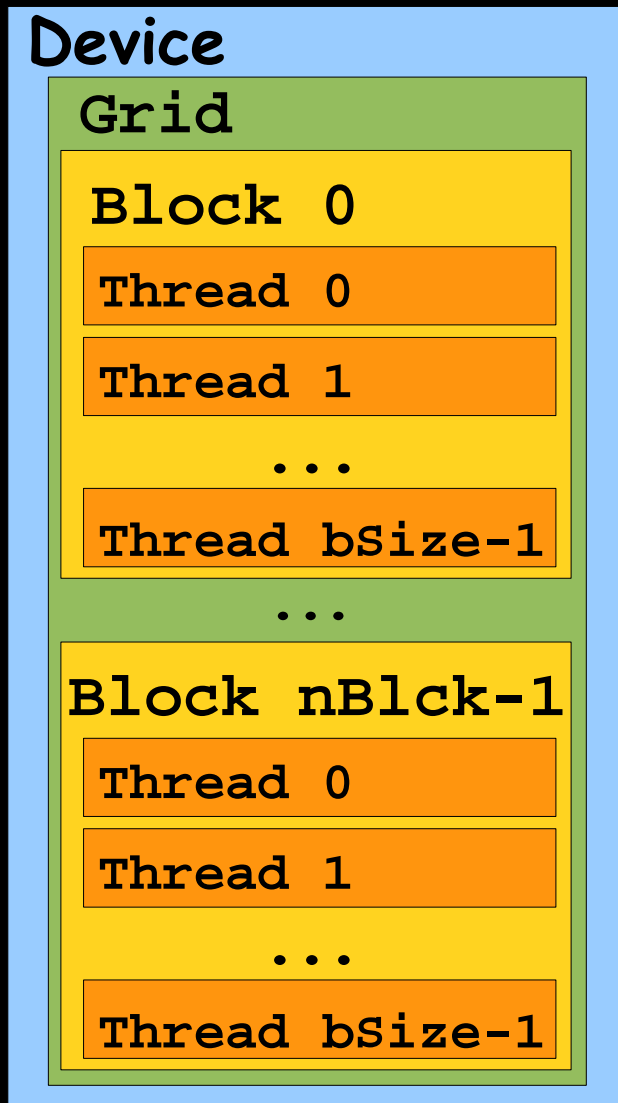
Fizikai felépítés



- N db **multiprocessor** (MP) ($N = 1..30..$)
- M db **singleprocessor**(SP) / MP
float: $M=8$ (double: $M=1$)
- Single Instruction Multiple Data (SIMD)
- Memória:
 - Global**: lassú, ~GB
 - Register**: gyors, 16384/Block
 - Shared**: gyors, 16kB

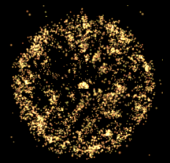


Futtatási modell

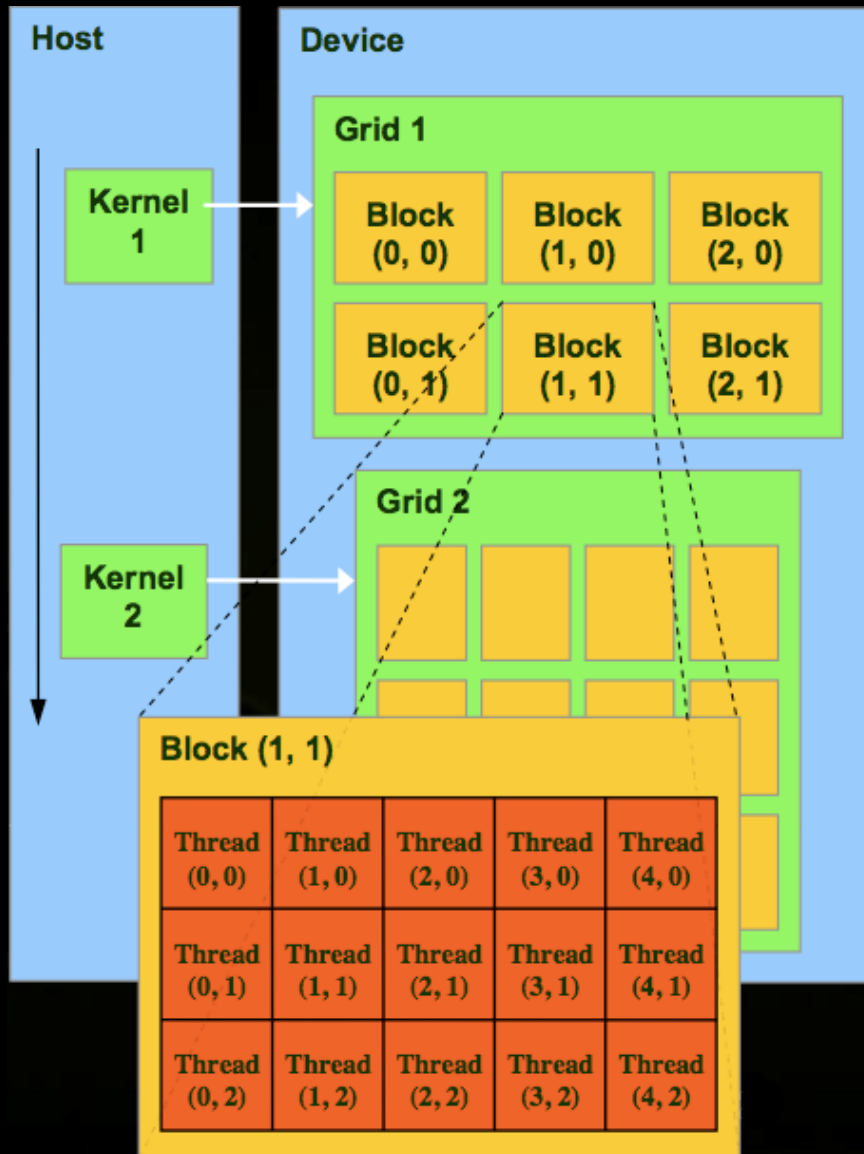


- **Kernel**: GPU-n futó objektum
- A **threadek** a kernelt hajtják végre
- A **grid thread blockokból** áll
- A blockok számát és méretét (execution configuration) a host alkalmazás állítja be
- Mindegyik thread azonosítja magát **blockIdx**, **threadIdx** (blockon belül), **blockDim**, **gridDim**

$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$



Futtatási modell



- Pl. 512 szál / Grid

blockDim.x	32	64
gridDim.x	16	8

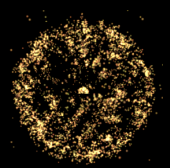
- Grid dim.: 1 vagy 2

- Block dim.: 1, 2 vagy 3

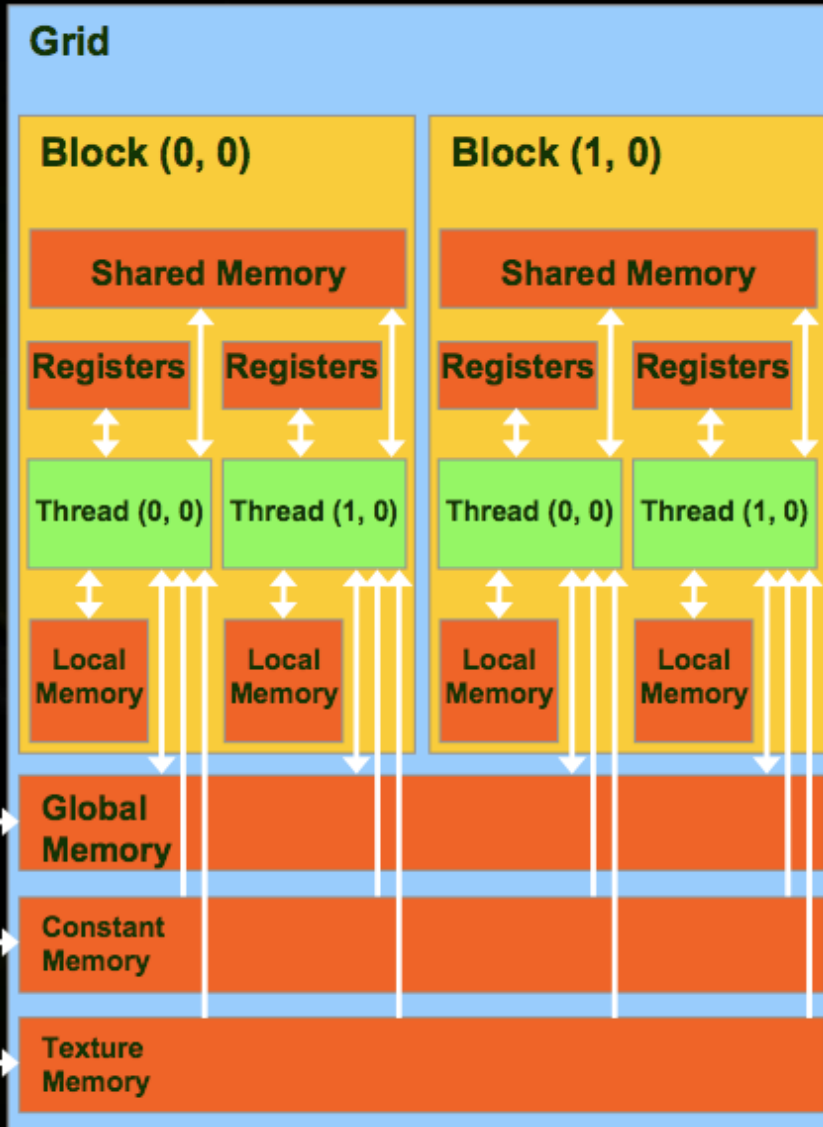
threadIdx., threadIdx.y,
threadIdx.z

- Pl. 64 szál / Block

blockDim.x	64	32
blockDim.y	1	2



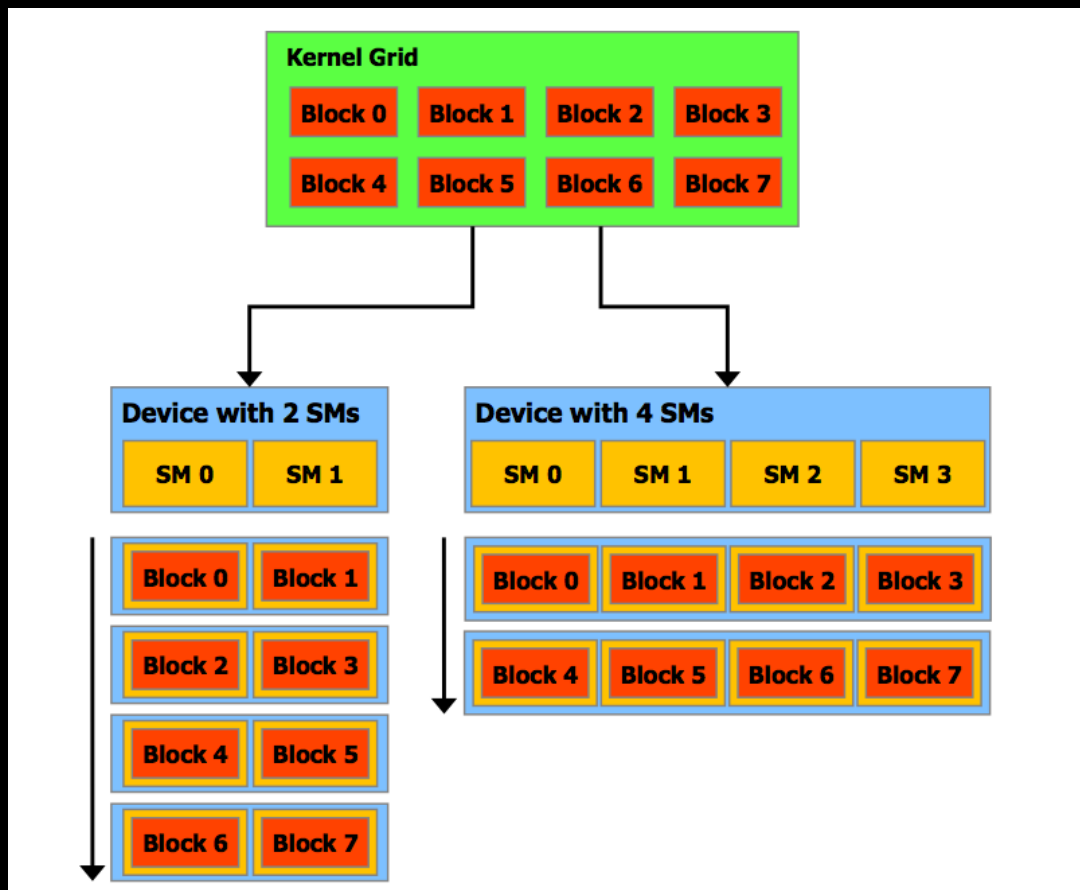
GPU memória

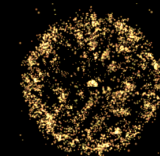


Mem.	láthatóság	sebesség
Global	grid	lassú
Shared	block	gyors
Register	thread	gyors

Fizikai végrehajtás

- Egy Blockot egy MP számol
- Minden Block SIMD csoportokra van osztva: **warp**
a warp threadjei fizikailag egyszerre hajtódnak végre
- a warp (ma) 32 százból áll (0-31, 32-63, ...)

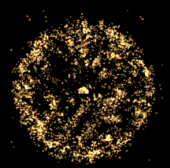




CUDA kódszerkezet

```
__device__ void mykernel(float *v, int D, float c) {...}

int main(void) {
    ...
    cudaMallocHost((void**) &h, s);
    cudaMalloc( (void**) &d, s) ;
    cudaMemcpy( d, h, s, cudaMemcpyHostToDevice);
    ...
    dim3 gridD      (30);
    dim3 blockD    (16,16);
    mykernel <<< gridD, blockD >>> (d, D, 2.0f);
    ...
    cudaMemcpy( h, d, s, cudaMemcpyDeviceToHost);
    cudaFreeHost(h);  cudaFree(d);
}
```



GPU memória

Memória - foglалás

Global: pl. host (CPU) kódrészben:

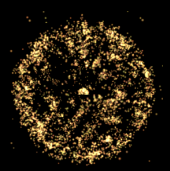
```
cudaMalloc( (void**) &g, memSize);
```

Register: a kernelben (`__device__` függvény)

pl. változó deklarálása: `float r;`

Shared: kernelben

```
pl. __shared__ float s[1000][4];
```



GPU memória

Memória - adattranszfer

Értékadással, pl.

- **Global -> Register**

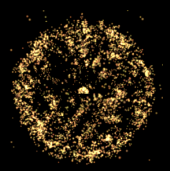
`r = g[2] ;`

- **Global -> Shared**

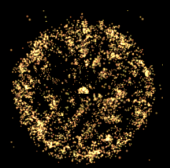
`s[0][0] = g[1] ;`

- **Shared -> Global**

`g[0] = s[0][0] ;`

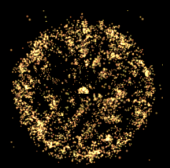


Tippek



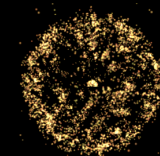
Tippek

- `nvcc` flagek:
 - emu : emuláció CPU-n (-> printf, debug)
 - keep : megőrzi a fordítás közbenső fájljait
 - arch sm_13 ; -arch compute_13 : double support
 - arch sm_10 ; -arch compute_10 : lekorlátozás
- Több végrehajtási konfiguráció (számítási háló, computational grid) kipróbálása (`gridDim`, `blockDim`)
- Hibakezelés (ld. *Supercomputing for the masses*)



Tippek

- Profiler (cudatoolkit)
- Debugger (cudatoolkit)
- Kártyalefagyás esetén:
 - kill -9 <application PID>
 - várni perceket
 - root-ként: rmmmod nvidia ; modprobe nvidia
- lib készítése (ld.: példaprogramok)

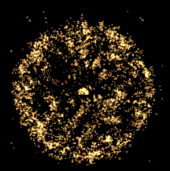


Tippek: időmérés

GPU-n:

```
cudaEvent_t start, stop;  
  
cudaEventCreate( &start ) ;  
cudaEventCreate( &stop ) ;  
  
cudaEventRecord( start, 0 ) ;  
  
{ ... }  
  
cudaEventRecord( stop, 0 ) ;  
  
cudaEventElapsedTime( &elapsedTimeInMs, start, stop );
```

ld. pl. `bandwidthTest.cu` forráskód (SDK)



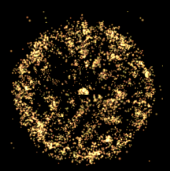
Tippek: időmérés

CPU-n:

```
struct timespec start, end;
clock_gettime(CLOCK_REALTIME, &start);
{ ... }
cudaThreadSynchronize(); //non-blocking GPU kernel!
clock_gettime(CLOCK_REALTIME, &stop);

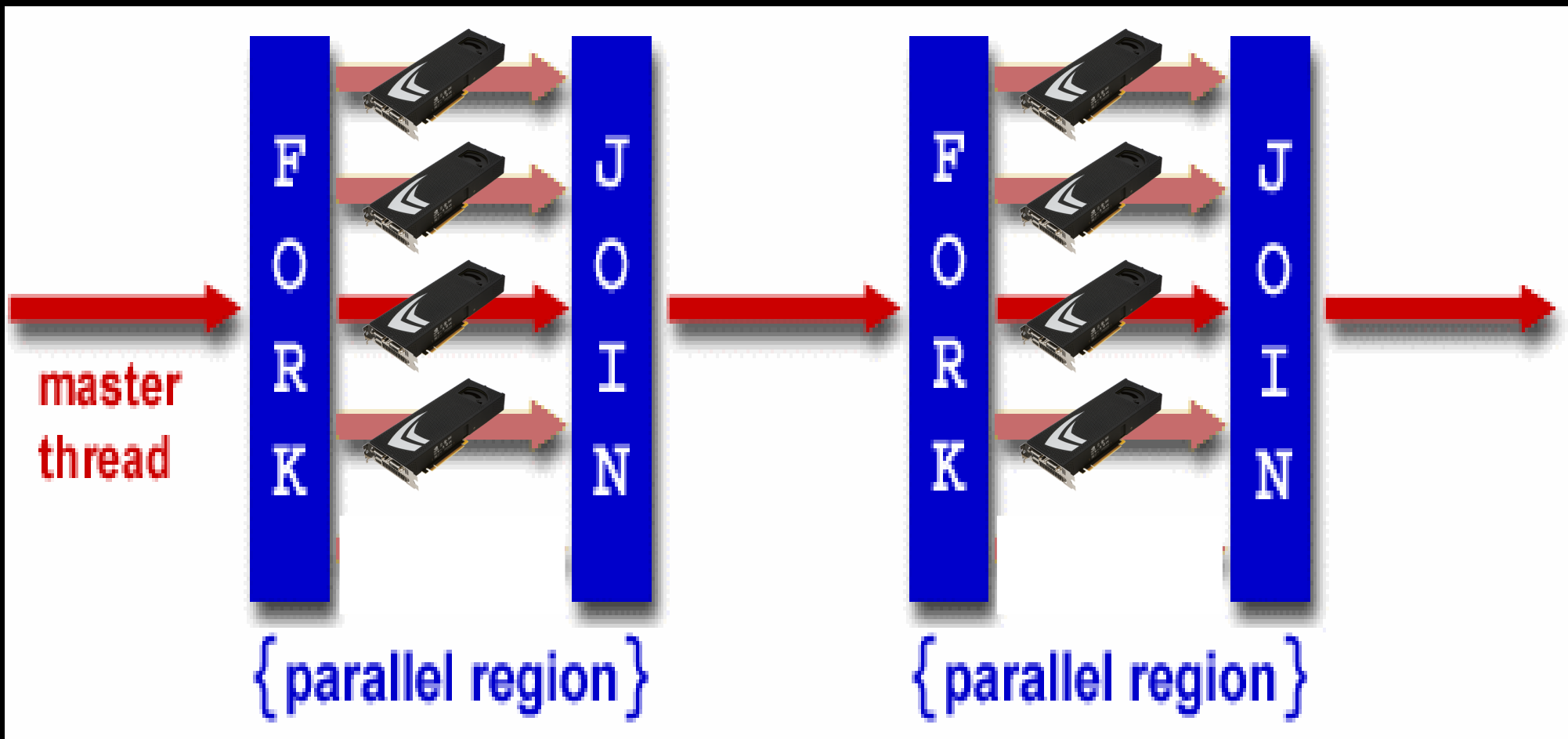
time_elapsed = (stop.tv_sec - start.tv_sec) +
(double)(stop.tv_nsec - start.tv_nsec) / 1000000000 );
```

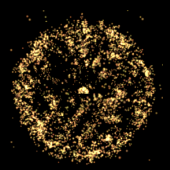
Megj: GPU kernel aszinkron végrehajtás ->
CPU/GPU párhuzamos számolás



Tippek: több kártya egy gépben

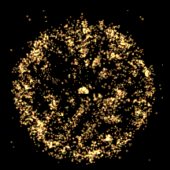
OpenMP + CUDA





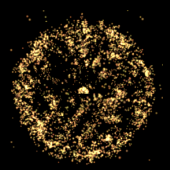
Tippek: több kártya egy gépben

```
int num_gpus, gpu_id, cpu_thread_id, num_cpu_threads;  
  
cudaGetDeviceCount(&num_gpus);  
omp_set_num_threads(num_gpus);  
  
#pragma omp parallel private(cpu_thread_id,gpu_id)  
{  
    cpu_thread_id = omp_get_thread_num();  
    num_cpu_threads = omp_get_num_threads();  
  
    cudaSetDevice(cpu_thread_id % num_gpus);  
    cudaGetDevice(&gpu_id);  
  
    dim3 gpu_threads(...);  
    dim3 gpu_blocks(...);  
    cuda_kernel <<< gpu_blocks,gpu_threads >>> (d,x);  
}
```

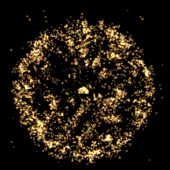


Portland CUDA fortran

<http://www.pgroup.com/resources/cudafortran.htm>



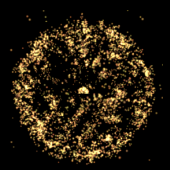
Könyvtárak



Könyvtárak

Optimalizált könyvtárak használata jelentősen felgyorsíthatja kódfejlesztésünket.

- Pl. - BLAS (Basic Linear Algebra Subprograms)
- FFTW (Fast Fourier Transform)



Könyvtárak

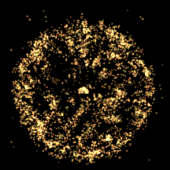
Optimalizált könyvtárak használata jelentősen felgyorsíthatja kódfejlesztésünket.

- Pl.
- BLAS (Basic Linear Algebra Subprograms)
 - FFTW (Fast Fourier Transform)

CUDA megvalósítás (Toolkit része):

- CUBLAS
- CUFFT

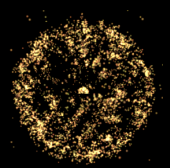
De jól át kell gondolni, hogyan használjuk!



PI. CUBLAS

Alapmodell

- Létrehozunk a vektor/mátrix objektumokat a GPU-n (host -> device adattranszfer)
- Meghívjuk a CUBLAS függvényeket (számolás GPU-n)
- Visszaolvassuk az eredményeket (device -> host adattranszfer)

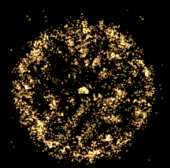


PI. CUBLAS

GPU számolás vs. Host-device adattranszfer

N elemű vektor (\underline{v}), $N \times N$ elemű mátrix (M) esetén:

	# aritm. műveletek	# adat- transzfer	#aritm/ átvitt adat
$c * \underline{v}, \underline{v} * \underline{v}$	$O(N)$	$O(N)$	$O(1)$
$M * \underline{v}$	$O(N^2)$	$O(N^2)$	$O(1)$
$M * M$	$O(N^3)$	$O(N^2)$	$O(N)$



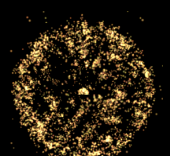
PI. CUBLAS

GPU számolás vs. Host-device adattranszfer

N elemű vektor (\underline{v}), $N \times N$ elemű mátrix (M) esetén:

	#aritm/ átvitt adat
$C \cdot \underline{v}, \underline{v} \cdot \underline{v}$	$O(1)$
$M \cdot \underline{v}$	$O(1)$
$M \cdot M$	$O(N)$

- $O(1)$: a teljesítmény erősen függ host→dev sávszélességtől (4GB/s → csak ~GFLOP)
- $O(N)$: jelentős gyorsulás érhető el

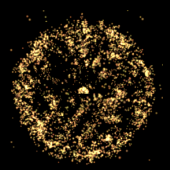


PI. CUBLAS

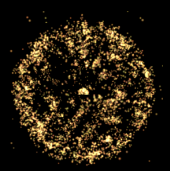
Összefoglalva:

- $O(1)$ aritm./transzfer esetén a cél minél tovább a GPU-n tartani az adatokat, minél több műveletet végezni rajta
- $O(N)$ aritm./transzfer esetén nagy gyorsulás várható; célszerű az $O(1)$ műveleteket is

Megj.: "Thunking" (auto mem. alloc, cpy + GPUcalc):
egyszerűbb használat, de jelentős lassulás (~2-4x)



GPU számolás CUDA programozás nélkül



Portland accelerator

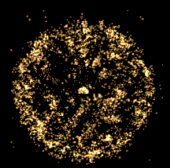
```
int main(void)
{
    ...
    {
        ...
    }
    ...
}
```

```
#include <acclmath.h>
int main(void)
{
    ...
    #pragma acc region
    {
        ...
    }
    ...
}
```

Portland accelerator

Fordítás:

```
pgcc sample.c -O3 -ta=nvidia -Minfo -fast
```

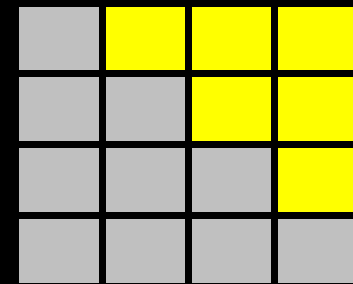


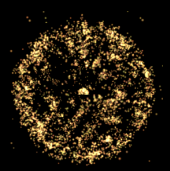
Portland accelerator

Regularizált gravitációs párkölsönhatás

```
#pragma acc region
```

```
{
  for (i=0; i<n; i=i+1) {
    for (j=i+1; j<n; j=j+1) {
      dx0 = xi[i][0] - xi[j][0];
      dx1 = xi[i][1] - xi[j][1];
      dx2 = xi[i][2] - xi[j][2];
      yyy = (dx0*dx0 + dx1*dx1) + (dx2*dx2+eps2) ;
      xxx = - m[i]*m[j] / ( sqrtf(yyy) * yyy ) ;
      ai[i][0] += xxx * dx0 ;
      ai[i][1] += xxx * dx1 ;
      ai[i][2] += xxx * dx2 ;
      aj[j][0] -= xxx * dx0 ;
      aj[j][1] -= xxx * dx1 ;
      aj[j][2] -= xxx * dx2 ;
    }
  }
}
```





Portland accelerator

Regularizált gravitációs párkölcsönhatás

Fordítási üzenet:

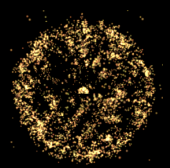
```
237, Accelerator restriction: size of the GPU copy  
of an array depends on values computed in this loop
```

```
238, Accelerator restriction: size of the GPU copy  
of 'm' is unknown
```

```
Accelerator restriction: size of the GPU copy  
of 'xi' is unknown
```

```
Accelerator restriction: one or more arrays  
have unknown size
```

```
Loop not vectorized: data dependency
```

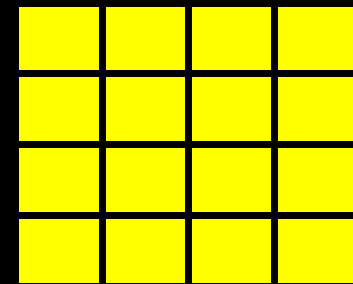


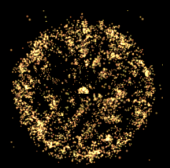
Portland accelerator

Regularizált gravitációs párkölsönhatás

```
#pragma acc region
```

```
{
  for (i=0; i<n; i=i+1) {
    for (j=0; j<n; j=j+1) {
      dx0 = xi[i][0] - xi[j][0];
      dx1 = xi[i][1] - xi[j][1];
      dx2 = xi[i][2] - xi[j][2];
      yyy = (dx0*dx0 + dx1*dx1) + (dx2*dx2+eps2) ;
      xxx = - m[i]*m[j] / ( sqrtf(yyy) * yyy ) ;
      ai[i][0] += xxx * dx0 ;
      ai[i][1] += xxx * dx1 ;
      ai[i][2] += xxx * dx2 ;
      // ai[j][0] -= xxx * dr[0] ;
      // ai[j][1] -= xxx * dr[1] ;
      // ai[j][2] -= xxx * dr[2] ;
    }
  }
}
```





Portland accelerator

Regularizált gravitációs párkölcsönhatás

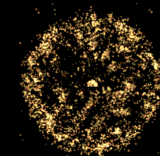
Fordítási üzenet:

```
235, Generating copyin(xi[0:natom-1][0:2])
      Generating copyin(m[0:natom-1])
      Generating copyout(ai[0:natom-1][0:2])
      Generating compute capability 1.0 kernel
      Generating compute capability 1.3 kernel
```

```
237, Loop is parallelizable
      Accelerator kernel generated
```

```
237, #pragma acc for parallel, vector(32)
      Non-stride-1 accesses for array 'xi'
      Non-stride-1 accesses for array 'ai'
```

```
244, Complex loop carried dependence of 'ai'
prevents parallelization
      Loop carried reuse of 'ai' prevents
parallelization
      Inner sequential loop scheduled on accelerator
```



Portland accelerator

Regularizált gravitációs párkölcsönhatás

Futásidők (N=65536)

	1 CPU	PGI acc. (trivial)	CUDA kód
Idők [s]	52	5 (~10x)	0.27 (~200x)

A CPU kód szervezésétől erősen függhet a GPU-s gyorsítás hatékonysága!

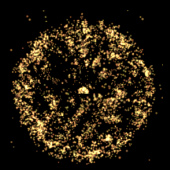
Irodalom pl.:

<http://www.pgroup.com/resources/accel.htm>

<http://www.pgroup.com/lit/articles/insider/v1n1a1.htm>

<http://www.pgroup.com/lit/articles/insider/v1n2a1.htm>

<http://www.pgroup.com/lit/articles/insider/v1n3a1.htm>



HMPP Workbench

a directive-based compiler for hybrid computing

SUPPORTED PLATFORMS AND COMPILERS

GPUs

- All NVIDIA Tesla
- NVIDIA CUDA compatible graphics products (GTX280, Quadro FX5800, GeForce 8800GTX, 9xx, ...)
- AMD FireStream 9170, 9250
- CAL compatible graphics products

Compilers

- GNU gcc 4.1 and above
- Intel icc version 9.1 and above
- Intel ifort version 9.1 and above

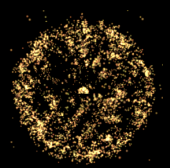
Operating systems

- Any x86_64 kernel 2.6 Linux distribution with libc coming with g++ 4.x and above.

- HMPP has been validated with some of the below Linux distributions:

- Debian 4.0 and above
- RedHat Enterprise Linux 5.x and above
- OpenSuse 11.x and above
- SLES 10.1, 10.2, 11.1
- Ubuntu 8.10

- **Windows**



Magasabb szintű nyelvek + CUDA

Természetes a CUDA kiterjesztés, ha lehetőség van C-ben írt modulok használatára

- Python - PyCUDA

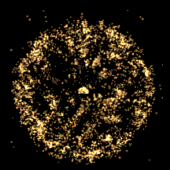
- Matlab

- mex -> nvmex

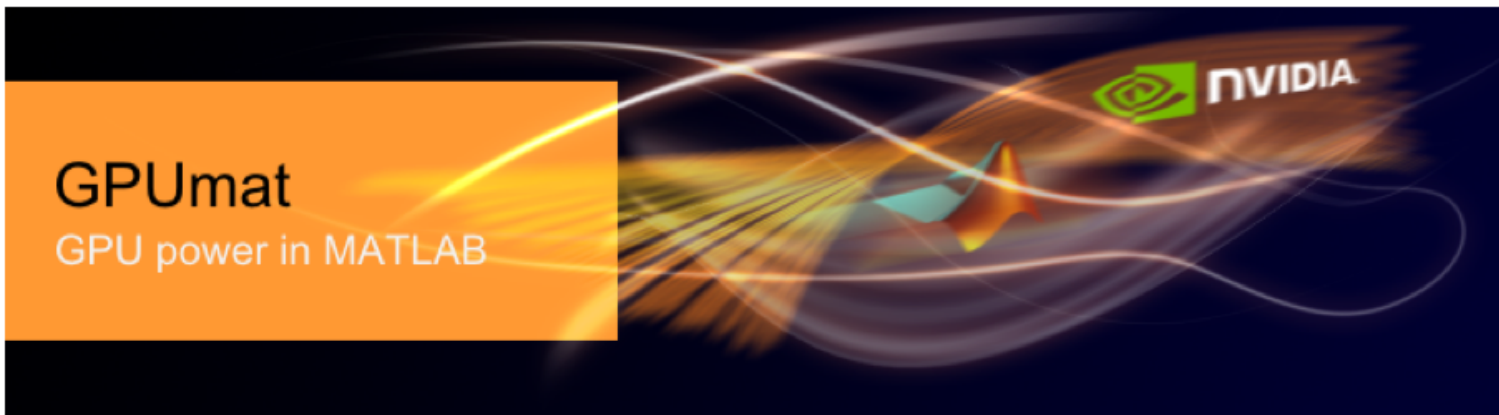
- http://developer.nvidia.com/object/matlab_cuda.html

- GPUmat / Jacket

Ügyelni kell a nyelvek közötti adatátvitel többletidejére!



Matlab + CUDA



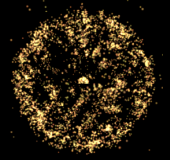
GPUmat allows standard MATLAB code to run on GPUs. The execution is transparent to the user as shown in the following example:

```
A = GPUsingle(rand(100)); % A is on GPU memory  
B = GPUDouble(rand(100)); % B is on GPU memory  
C = A+B; % executed on GPU.  
D = fft(C); % executed on GPU
```

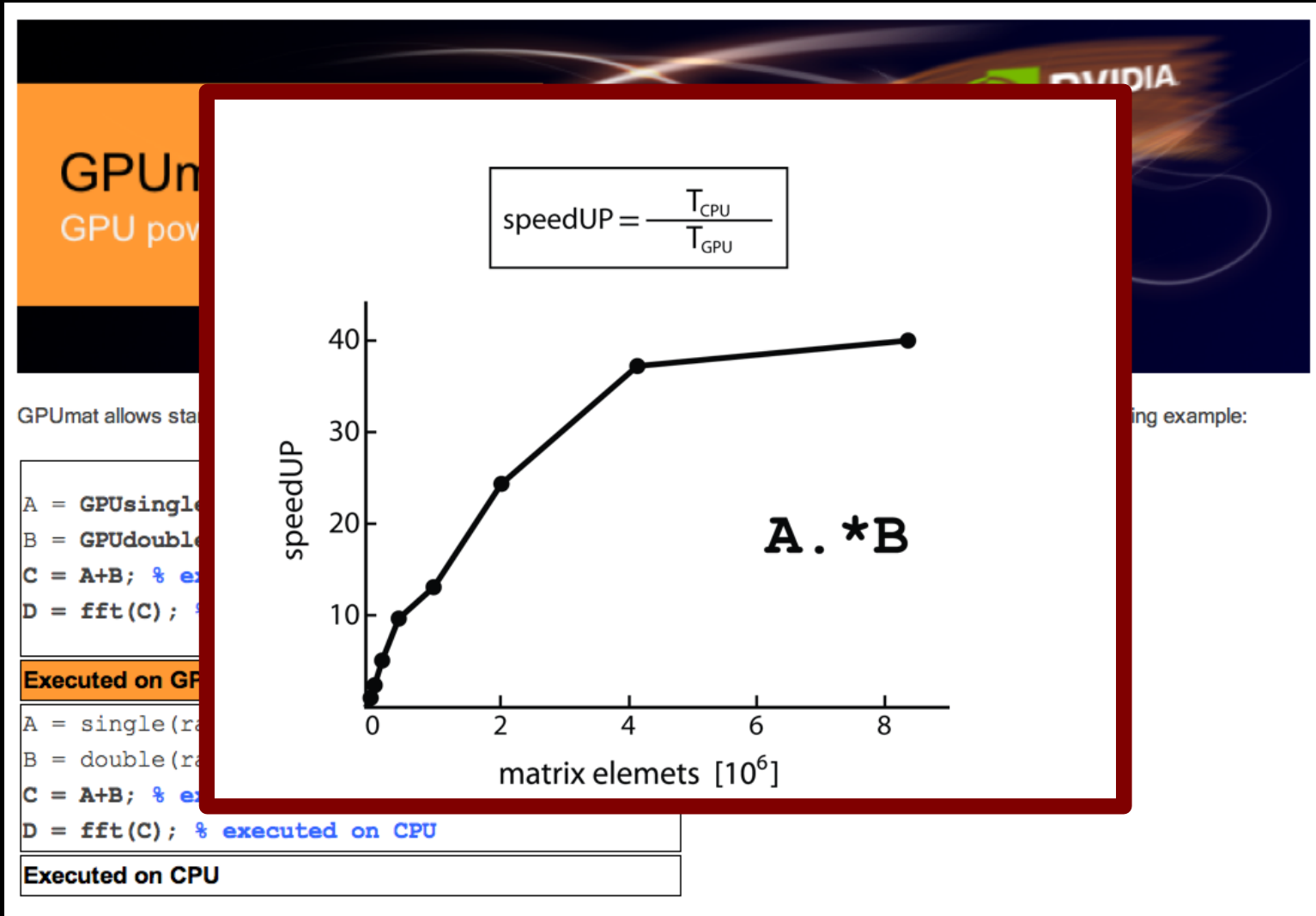
Executed on GPU

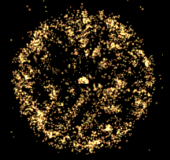
```
A = single(rand(100)); % A is on CPU memory  
B = double(rand(100)); % B is on CPU memory  
C = A+B; % executed on CPU.  
D = fft(C); % executed on CPU
```

Executed on CPU



Matlab + CUDA





Újdonság: nVidia Fermi

<http://developer.nvidia.com/object/gpucomputing.html>

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit