

Heterogén számítási rendszerek

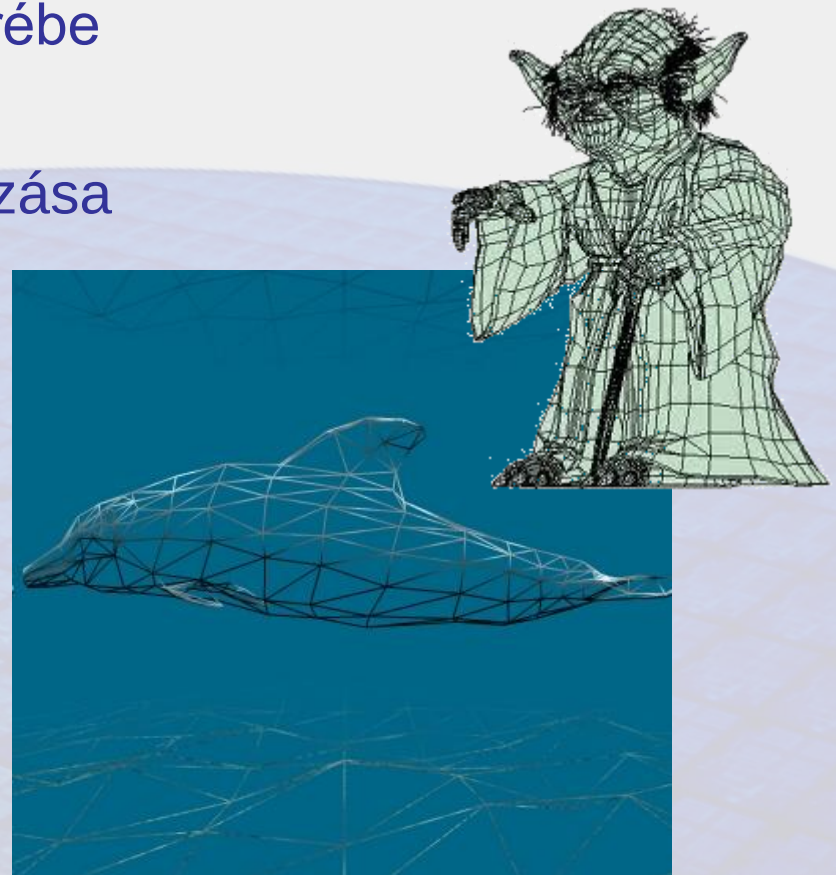
Bevezető

Szántó Péter

BME MIT, FPGA Laboratórium

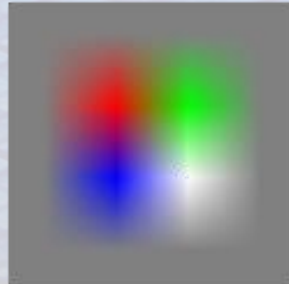
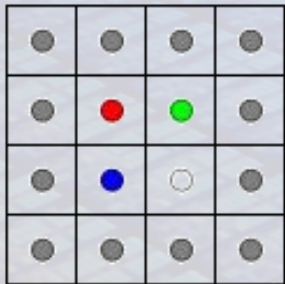
GPU-k

- **Graphics Processing Unit**
- **2 fő feladat**
 - Objektumok transzformációja a lokális koordináta rendszerből a képernyő koordináta rendszerébe
 - Raszterizáció
 - látható objektum meghatározása
 - pixelek színének számítása
- **Transzformáció**
 - Objektumok: háromszögekből
 - Transzformáció: homogén koordinátákban – X, Y, Z, W
 - Tipikus: forgatás, skálázás, perspektív vetítés



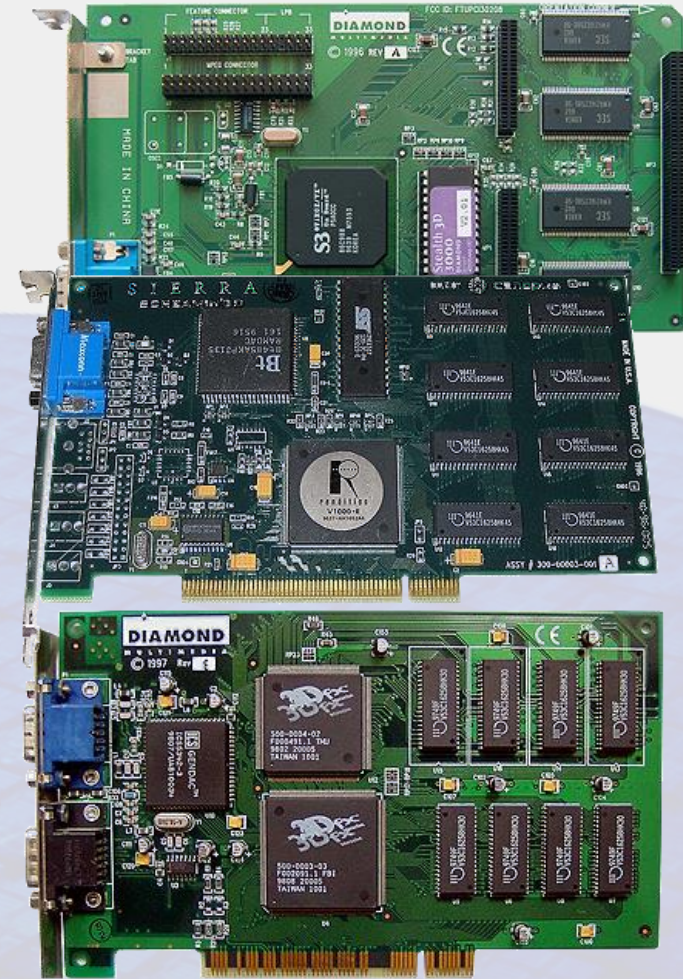
GPU – pixel színének számítása

- Színinformáció 4 csatornán: RGBA
- Tipikusan SP lebegőpontos számítás csatornánként
- Háromszög csúcspontok színének lineáris interpolációja
- Textúra leképzés
 - „kép ráfeszítése” az objektum háromszögre
 - lokális, jól cache-elhető hozzáférés
 - bilineáris szűrés dedikált HW-rel



GPU-k: a kezdetek

- **1995: S3 Virge, ATI Rage 3D/Rage II**
 - csak fix funkciós raszterizáció
 - 66 MHz, 4 MB RAM
 - „grafikus lassító”
- **1996**
 - 3Dfx Voodoo
 - csak 3D
 - 50 MHz, 4 MB
 - Saját API: Glide
 - Rendition Verite V1000
 - 25 MHz, 4 MB
 - DSP jellegű (?)



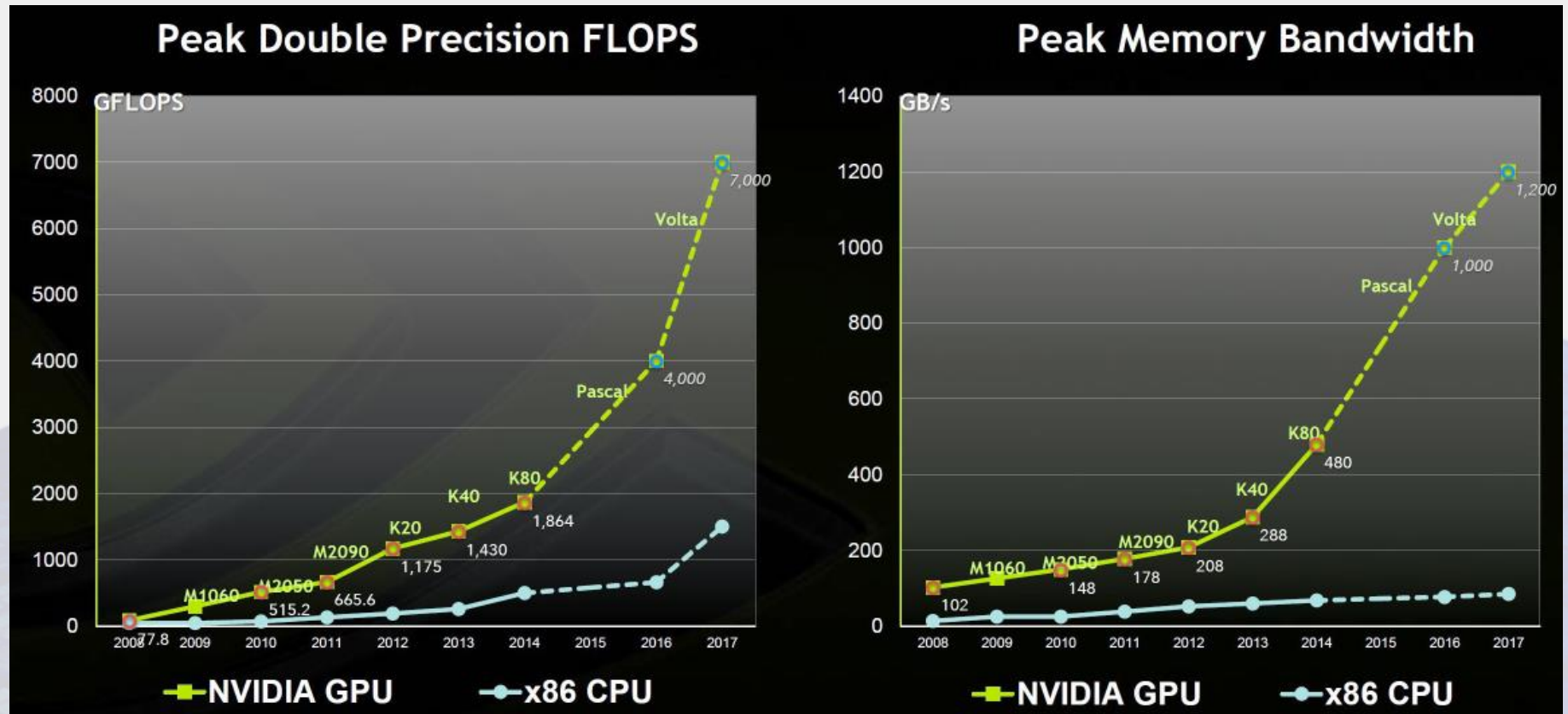
GPU-k: CUDA megjelenése

- **2001: Programozható shader-ek**
 - NVIDIA Geforce 3
 - ATI Radeon 8500
- **2002: Lebegőpontos feldolgozók (vertex & pixel shader)**
 - ATI Radeon 9700
 - NVIDIA GeForce FX
- **2006: „Unified Shader”**
 - NVIDIA G80 (GeForce 8)
 - Az első CUDA képes GPU
 - ATI Radeon 2000



CPU – GPU elméleti sebességek

- Miért lehet gyorsabb a GPU



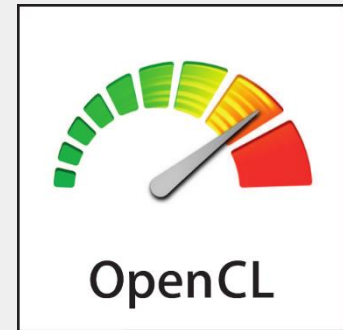
GPU programozás: grafikus API-k

- **Grafikus API-k**
 - Nem volt cél az általános célú felhasználás
 - Meg kell küzdeni a grafika „overhead”-del
- **Címzési módok**
 - Limitált textúra méret
- **Shader-ek**
 - Kimenetek száma, helye limitált
- **Utasításkészlet**
 - Nem teljes (int, bitműveletek)
- **Limitált kommunikáció a szálak (pixelek) között**

GPU programozás: GPGPU API-k

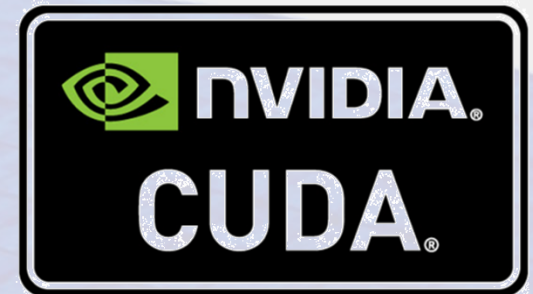
- **OpenCL**

- Több platformot támogat
 - GPU-k, CPU-k, FPGA-k
- Gyártó független
 - AMD, ARM, NVIDIA, Qualcomm



- **CUDA**

- Az NVIDIA saját megoldása
 - HPC iparban a legelterjedtebb
- AMD HIPify: AMD fordító CUDA-hoz
- (GPU Ocelot: futtató környezet CPU-n)



- **Egyebek**

- Microsoft DirectCompute
- C++ AMP
- OpenACC
- Intel One API

OpenCL

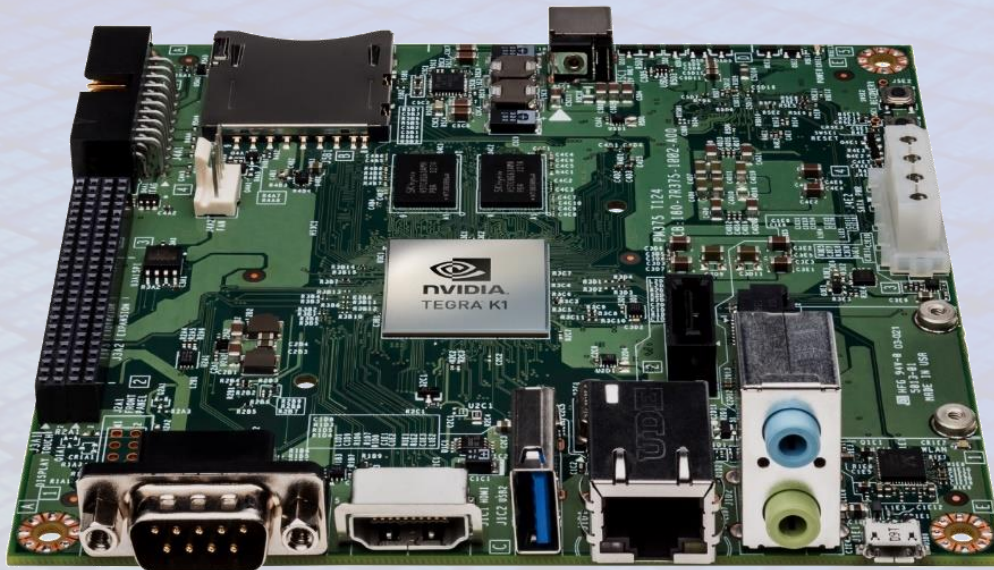
- **Eszköz és operációs rendszer független**
- **AMD APP SDK (<https://gpuopen.com/>)**
 - OpenCL 2.1 driver
 - CPU és AMD GPU driver
 - AMD CodeXL debugger: kernel debug GPU-n
- **Intel OpenCL SDK**
 - OpenCL 2.1 driver CPU-hoz
 - OpenCL 3.0 driver HD Graphics GPU-khoz
 - Kernel debug
- **NVIDIA**
 - OpenCL 3.0 driver (csak x64); kb. 0 fejlesztői támogatás
 - Valójában OpenCL 1.2 néhány kiegészítéssel

Beágyazott eszközökben (1.)

- **OpenCL 1.0 Embedded Profile**
 - Enyhébb követelmények (pl. számítási precizitás)
- **Ma már beágyazott környezetben is elterjedt a teljes OpenCL 1.x/2.0 támogatás**
- **Operációs rendszer támogatás: iOS, Android**
 - Kompatibilis GPU-k (jobbára full profile)
 - ARM Mali T6xx és újabbak (Midgard: 1.2, újabbak 2.0)
 - Imagination PowerVR Series 5-től (1.2)
 - Qualcomm Adreno 3xx/4xx/5xx/6xx (1.2, 6xx-től 2.0)
 - Vivante (1.2)
- **Android: vegyes, de sok gyártónál van támogatás**
- **Linux támogatás khmmm....**

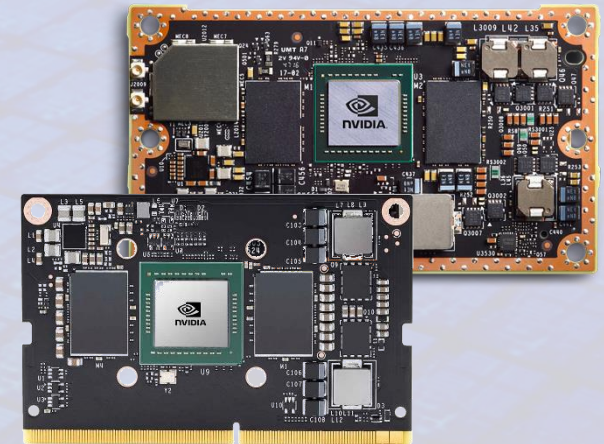
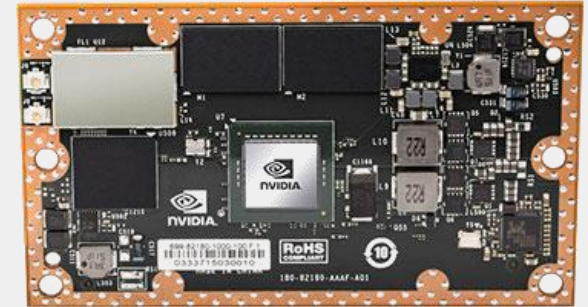
Beágyazott eszközökben (2.)

- **NVIDIA Tegra K1**
 - 4x ARM Cortex A15 CPU @ 2,3 GHz
 - 2x32 bit DDR3 memória @ 1866 Mb/s (15 GB/s)
 - K1 Kepler GPU: 192 mag @ 950 MHz
- **Jetson TK1 kártya**
- **Shield Tablet, Chromebook-ok**



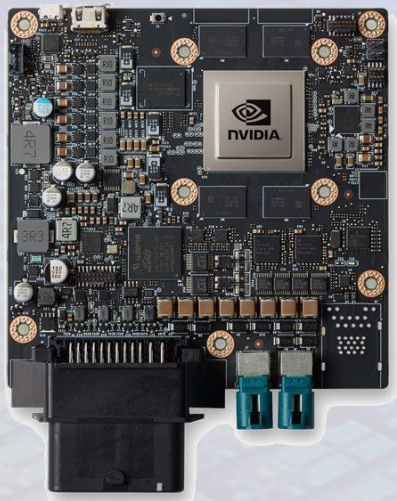
Beágyazott eszközökben (3.)

- **NVIDIA Jetson X1**
 - 4x ARM Cortex A57 @ 1,91 GHz
 - 2x32 bit LPDDR4 @ 3200 Mb/s (25,6 GB/s)
 - X1 Maxwell GPU: 256 mag, 1000 MHz
 - 4k@30 H265 encoder
- **NVIDIA Jetson NANO (butított X1)**
 - 4x ARM Cortex A57 @ 1,43 GHz
 - 2x32 bit LPDDR4 @ 3200 Mb/s (25,6 GB/s)
 - 128 magos GPU
- **NVIDIA Jetson TX2, TX2 NX**
 - 2x NVIDIA Denver + 4x ARM Cortex A57
 - 2x64 bit LPDDR4 @ 1866 Mb/s (59,7 GB/s)
 - 256 magos Pascal GPU
 - 4k@60 H265 encoder



NVIDIA Drive PX2

- „Megalomán” verzió
 - 2x Tegra X2
 - 2x Denver + 4x ARM A57
 - 256 ALU-s Pascal GPU
 - Pascal diszkrét GPU
 - 8 TFLOPS, 250W
- SoC verzió



NVIDIA Jetson AGX Xavier

- **Elsősorban autóiparba szánt SoC**
 - 8 magos 64 bites ARM CPU, 8 MB L2 + 4 MB L3 cache
 - 512 magos Volta GPU
 - VLIW Vision Processor
 - Deep learning gyorsító
 - 2x 4K 60p H264/H265 dekóder és encoder
 - 16 GB LPDDR4 RAM (137 GB/s sávszélesség)
 - PCIe 4.0, USB 3.1, GigETH
 - 16-lane MIPI kamera interfész
 - ~35W TDP, \$1000



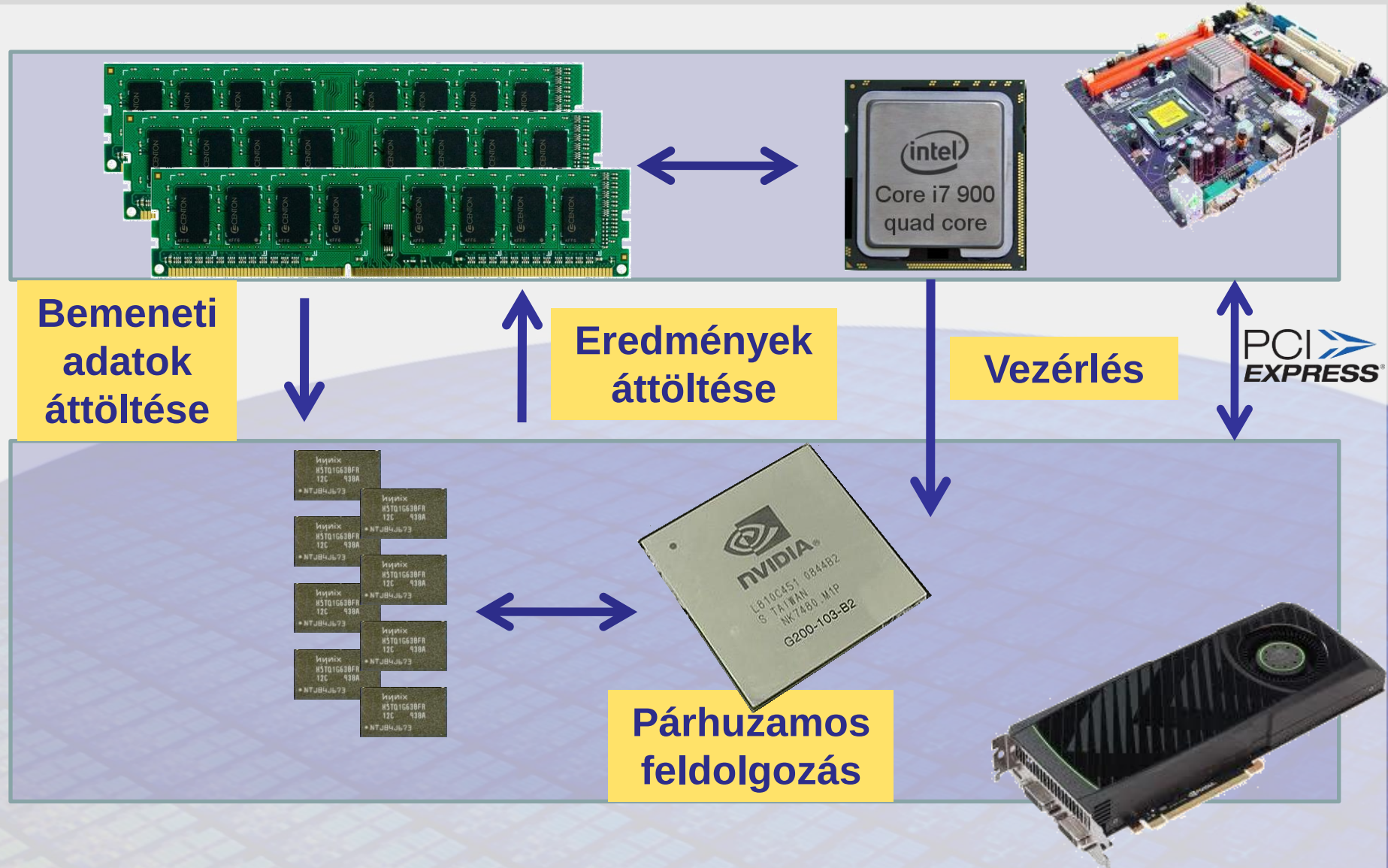
Heterogén számítási rendszerek

CUDA alapok

Szántó Péter

BME MIT, FPGA Laboratórium

Feldolgozási modell



CUDA: verziók

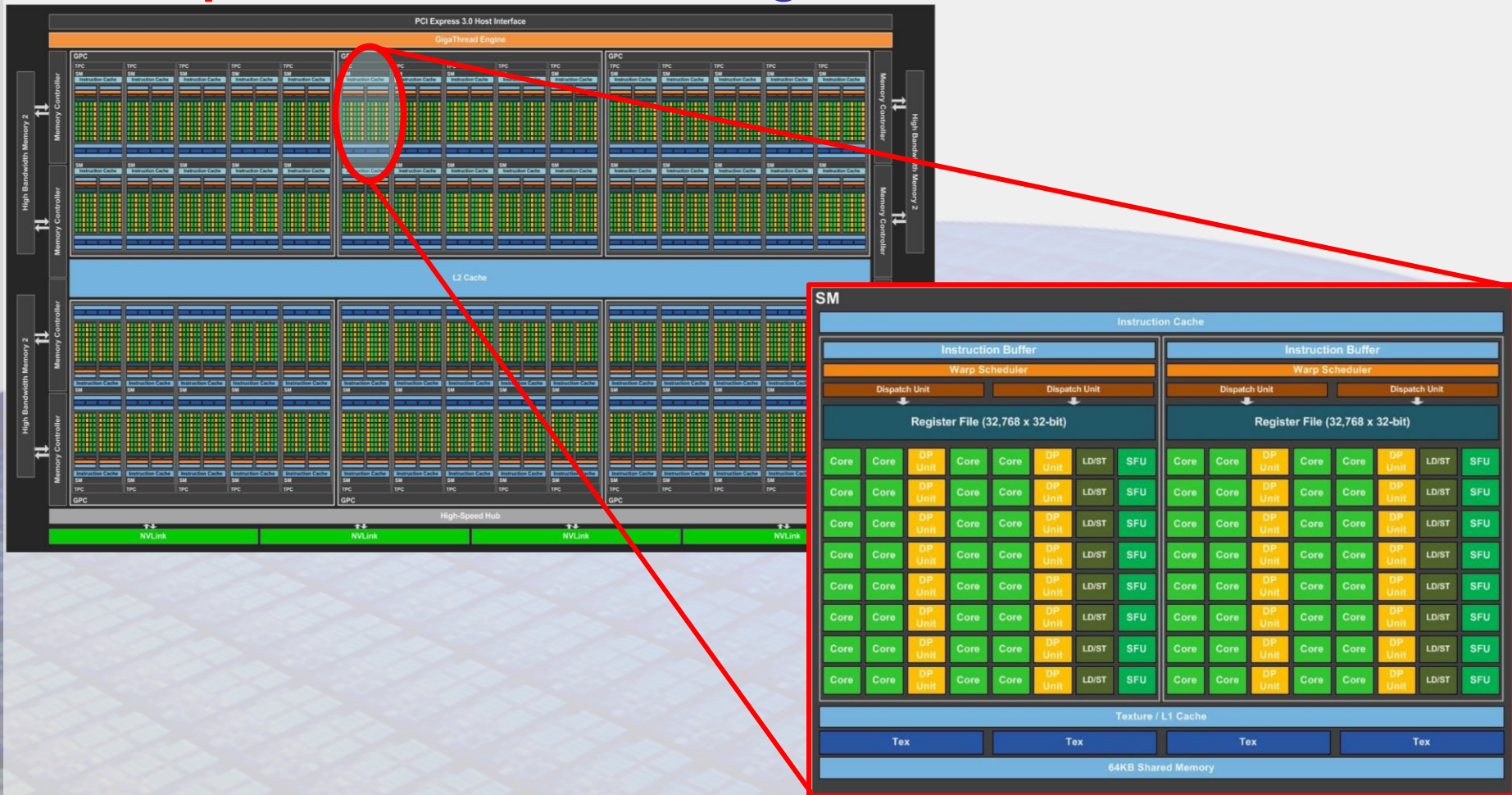
- **CUDA Toolkit verzió**
 - SW verziószám, semmi köze a HW-hez
 - Fermi csak 8.0-ig támogatott
- **Compute Capability: HW képességek verziószáma**
 - 1.x: Fermi előtti architektúrák
 - 2.0: GF100 Fermi
 - 2.1: GF100 leszármazottak (GF104, GF108)
 - 3.0: GK104 és társai (3.2: Tegra K1)
 - 3.5: GK110
 - 5.x: Maxwell (5.3: Tegra TX1)
 - 6.x: Pascal (6.2: Tegra TX2)
 - 7.x: Volta (7.2: AGX Xavier)
 - 7.5: Turing
 - 8.x: Ampere

CUDA

- **C nyelv minimális kiterjesztése**
 - Kernel: a GPU-n futó kód
 - Nincs rekurzió
 - Nincs dinamikus memória foglalás
 - Hoszt: a CPU-n futó kód
 - Driver API: viszonylag alacsony szintű interfész, JIT
 - Runtime API: a Driver API-ra épülő kényelmes programozási felület
- **Mindkét kód .cu file-ban**
 - nvcc fordítja a CUDA specifikus részeket
 - C/C++ kódot továbbadja a C/C++ fordítónak

GPU hardver hierarchia

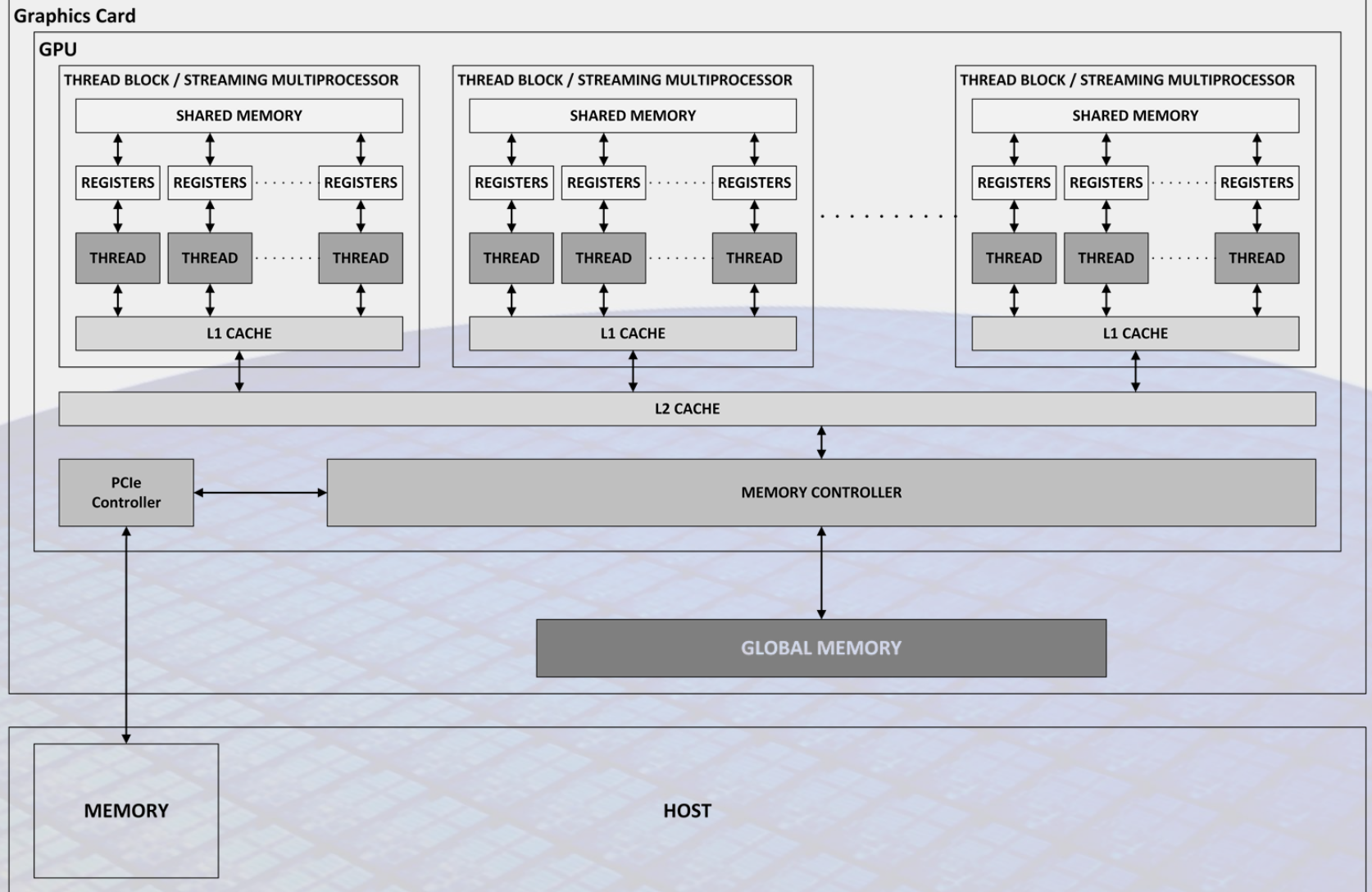
- Chip → Graphics Processing Cluster → **Streaming Multiprocessor** → ALU-k, regiszter, belső memória, stb.



CUDA szál hierarchia

- **Thread (work item):** szál, a kernel kód egy példánya
- **Warp:** 32 szál, ugyanazt az utasítást futtatják időosztásban egy Streaming Multiprocessor-on (SM)
 - Divergens szál: mindkét ág végrehajtása!
- **Thread Block (work group):** garantáltan egy SM-en futó szálak tömbje
 - Adatcsere
 - Szinkronizáció
- **Thread Grid:** Thread Block-okból álló tömb
 - Egymástól gyakorlatilag teljesen független

GPU memória hierarchia



Kernel futtatás

- Thread Block mérete és a Thread Block-ok száma a kernel indításánál dől el
 - dim3: beépített változó típus, legfeljebb 3 eleme van (x,y,z) – a nem használt 1
- Pl. 16x16 szálak Thread Block 8x8-as grid-ben → összesen $(16*16)*(8*8)=16384$ szál

```
dim3 thrBlock(16, 16);
```

```
dim3 thrGrid(8, 8);
```

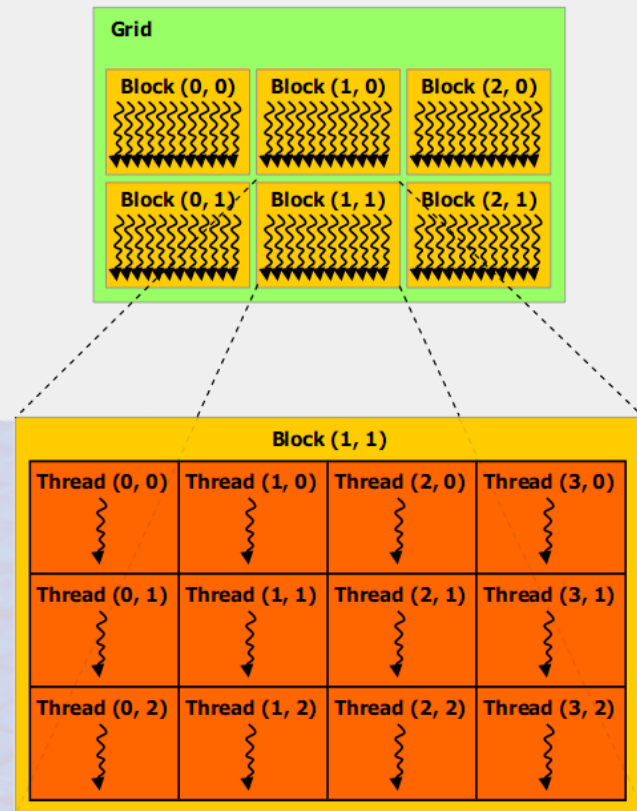
```
cudaKernel<<<thrGrid, thrBlock>>>( ..... );
```

- Thread Block méret: warp többszörös, praktikus 128 - 512

Kernel – szál azonosítók

- **threadIdx (.x, .y, .z)**
 - Egy Thread Block-on belül azonosítja a szálát
- **blockIdx (.x, .y, .z)**
 - A Thread Block sorszáma
- **blockDim (.x, .y, .z)**
 - Thread Block mérete
- **gridDim (.x, .y, .z)**
 - Thread Block-ból álló rács mérete
- Pl. 1D esetben egy szál abszolút sorszáma:

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



Vektor összeadás – CUDA kernel

- Kernel visszatérési értéke: mindig void
- A, B, C: a videokártya memóriájában foglalt memóriaterület pointere
- N: a vector méret

```
__global__ void VecAdd(float* A, float* B, float* C,  
    int N)  
{  
    int thID = blockDim.x * blockIdx.x + threadIdx.x;  
    if (thID < N)  
        C[thID] = A[thID] + B[thID];  
}
```

Vektor összeadás – CUDA hoszt

```
#define N 65536
int main()
{
    float *hA, *hB, *hC;
    hA = (float*) (malloc(N*sizeof(float)));
    hB = (float*) (malloc(N*sizeof(float)));
    hC = (float*) (malloc(N*sizeof(float)));

    float *dA, *dB, *dC;
    cudaMalloc((void**) &dA, N*sizeof(float));
    cudaMalloc((void**) &dB, N*sizeof(float));
    cudaMalloc((void**) &dC, N*sizeof(float));

    .....
}
```

**Hoszt memória
foglalása**

**GPU memória
foglalása**

Memória másolás – CUDA

**Bemeneti
adatok
másolása**

```
.....  
  
int size = N*sizeof(float);  
cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);  
cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);
```

**Kernel
futtatása**

```
int thrBlock = 256;  
int thrGrid =  
    int(ceil(float(N)/float(thrBlock)));  
VecAdd<<<thrGrid, thrBlock>>>(dA, dB, dC);
```

```
cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);  
cudaFree(dA); cudaFree(dB);  
cudaFree(dC);
```

**Eredmények
másolása**

```
}
```

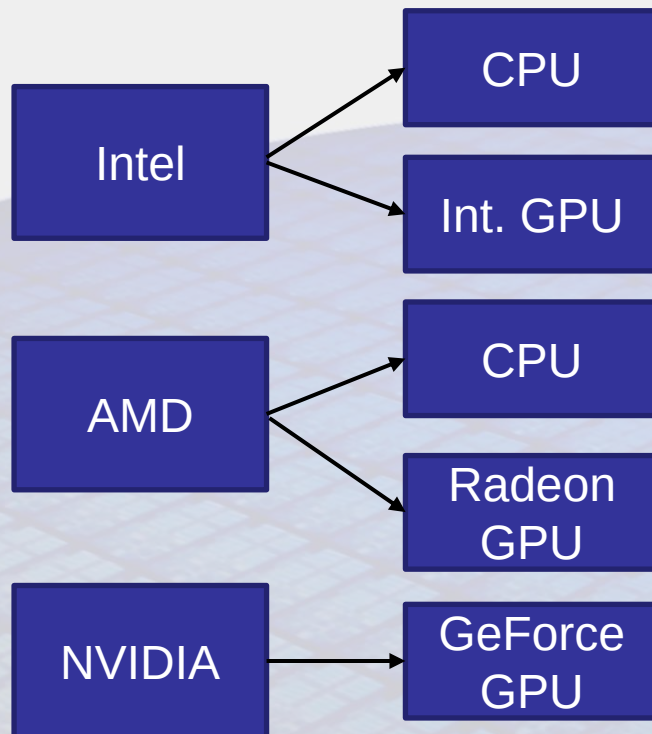
Vektor összeadás – OpenCL kernel

- Kernel visszatérési értéke: mindig void
- A, B, C: a videókártya memóriájában foglalt memóriaterület pointere
- N: a vektor méret

```
__kernel void VecAdd(__global float* A, __global float*  
    B, __global float* C, int N)  
{  
    int thID = get_global_id(0);  
    if (thID < N)  
        C[thID] = A[thID] + B[thID];  
}
```

OpenCL platform & eszköz

- Az egyes eszközök drivere egy ICD (installable client driver)
- Egy ICD → egy platform (lehet több)
- Egy ICD-n belül több eszköz → device
- Pl. Intel CPU-s gépben AMD és NVIDIA GPU:



Vektor összeadás – OpenCL hoszt

```
int N = 65536; // Vector size
float *hA, *hB, *hC; // Host memory pointers
cl_mem dA, dB, dC; // Device memory pointers
cl_platform_id cpPlatform; // OpenCL platform
cl_device_id device_id; // device ID
cl_context context; // context
cl_command_queue queue; // command queue
cl_program program; // program
cl_kernel kernel; // kernel
size_t globalSize, localSize;
cl_int err;
char *kernel_source; // kernel source code

int bytes = N*sizeof(float);

hA = (float*)malloc(bytes);
hB = (float*)malloc(bytes);
hC = (float*)malloc(bytes);
```

Vektor összeadás – OpenCL hoszt

```
localSize = 256;
globalSize = ceil(float(N)/(float)localSize)*localSize;

err = clGetPlatformIDs(1, &cpPlatform, NULL);
err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id,
    NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
queue = clCreateCommandQueue(context, device_id, 0, &err);

program = clCreateProgramWithSource(context, 1, (const char **)
    &kernelSource, NULL, &err);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "vecAdd", &err);

dA = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
dB = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
dC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
.....
```

Vektor összeadás – OpenCL hoszt

```
.....  
err = clEnqueueWriteBuffer(queue, dA, CL_TRUE, 0, bytes, hA, 0,  
    NULL, NULL);  
err |= clEnqueueWriteBuffer(queue, dB, CL_TRUE, 0, bytes, hB, 0,  
    NULL, NULL);  
  
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dA);  
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &dB);  
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dC);  
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &N);  
  
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,  
    &localSize, 0, NULL, NULL);  
  
clFinish(queue);  
  
clEnqueueReadBuffer(queue, hC, CL_TRUE, 0, bytes, dC, 0, NULL, NULL  
    );
```

Heterogén számítási rendszerek

CUDA: részletek

Szántó Péter

BME MIT, FPGA Laboratórium

NVIDIA GF100 (Fermi)

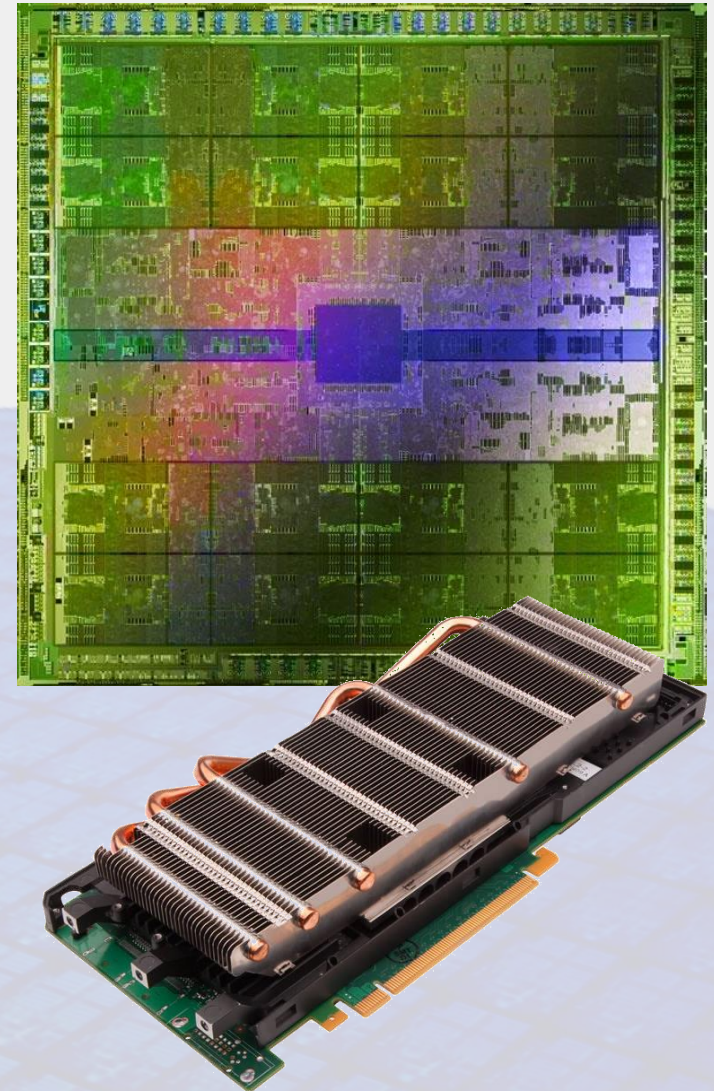
- **NVIDIA GF100 chip (Tesla M2090)**

- 512 ALU
- 3 milliárd tranzisztor
- 650/1300 MHz
- 1,33 TFLOPS SP MAD
- 0,655 TFLOPS DP MAD

- **GDDR5 memória**

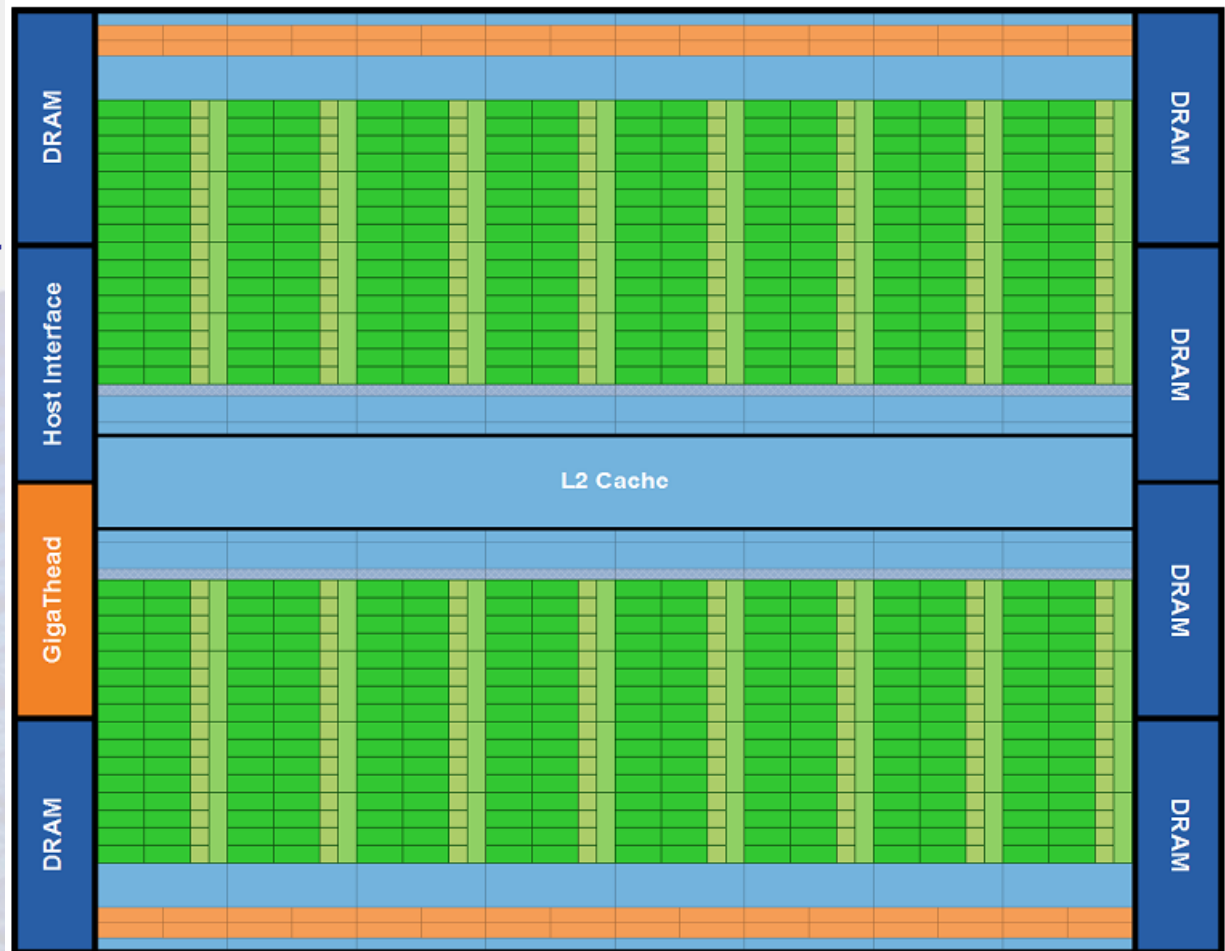
- 384 bites busz
- ECC támogatás
- 3700 MHz
- ~178 Gbyte/s sávszélesség

- **250 W fogyasztás**



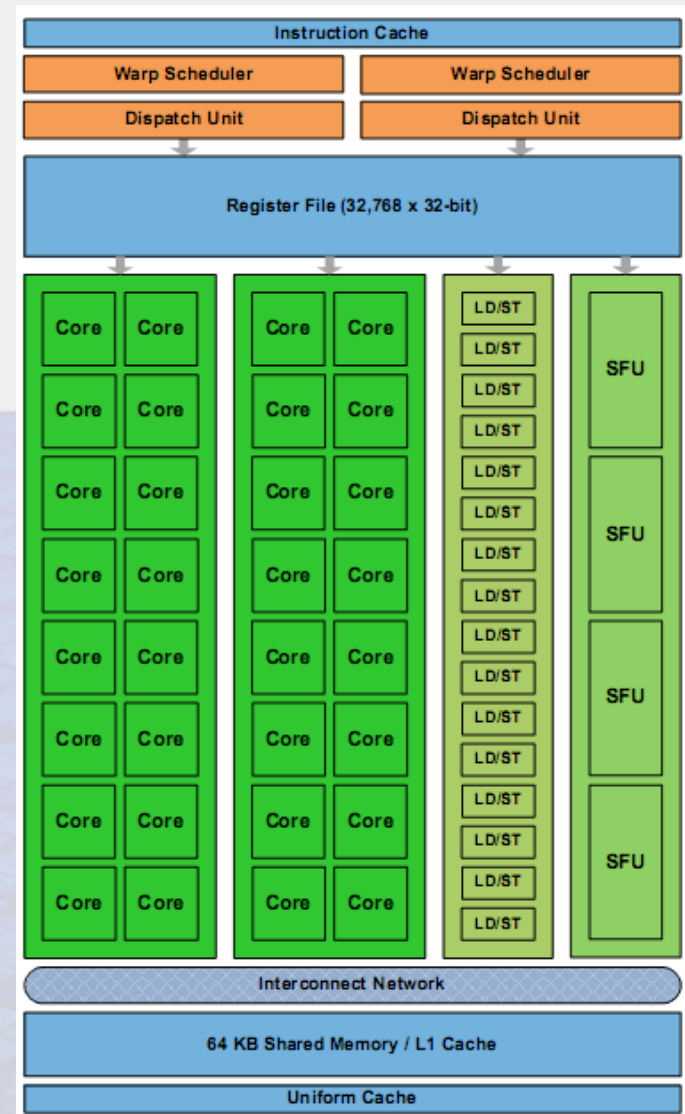
NVIDIA GF100 architektúra

- 16 Streaming Multiprocessor (SM)
- 32 SP / SM
- 768 kbyte L2 cache
- 6x64 bit memória vezérlő (ECC)
- 64 bit címzés



NVIDIA GF100 SM

	GF100
ALU	32
SFU	4
Regiszter	32k*32
Shared Memory	16 – 48 kbyte
Cache	16 – 48 kbyte
Threads	1536



NVIDIA GF100 SM

- **FPU**

- DP: 2 clk/DP utasítás
- SP & DP FMA (fused multiply-add) támogatása → nincs kerekítés a szorzás után

- **ALU**

- 32 bites szorzás
- rész-kiválasztás, 1 számlálás, bit sorrend fordítás, ...

- **2 warp párhuzamos feldolgozása**

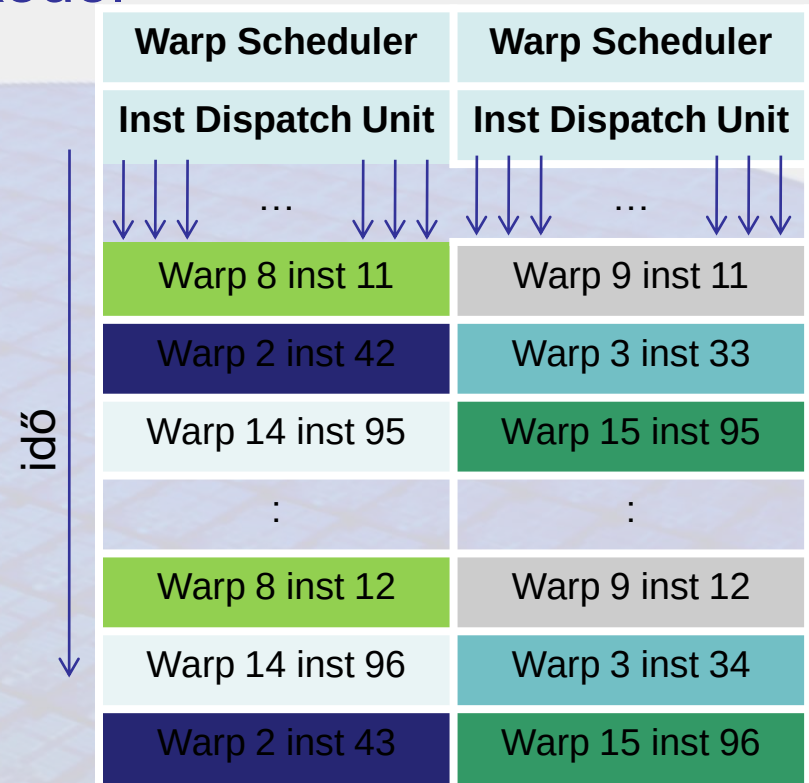
- 16 – 16 feldolgozó egységen

- **64 kbyte memória**

- 16 kbyte Shared Memory & 48 kbyte L1 cache vagy
- 48 kbyte Shared Memory & 16 kbyte L1 cache

GF100 – Warp ütemező

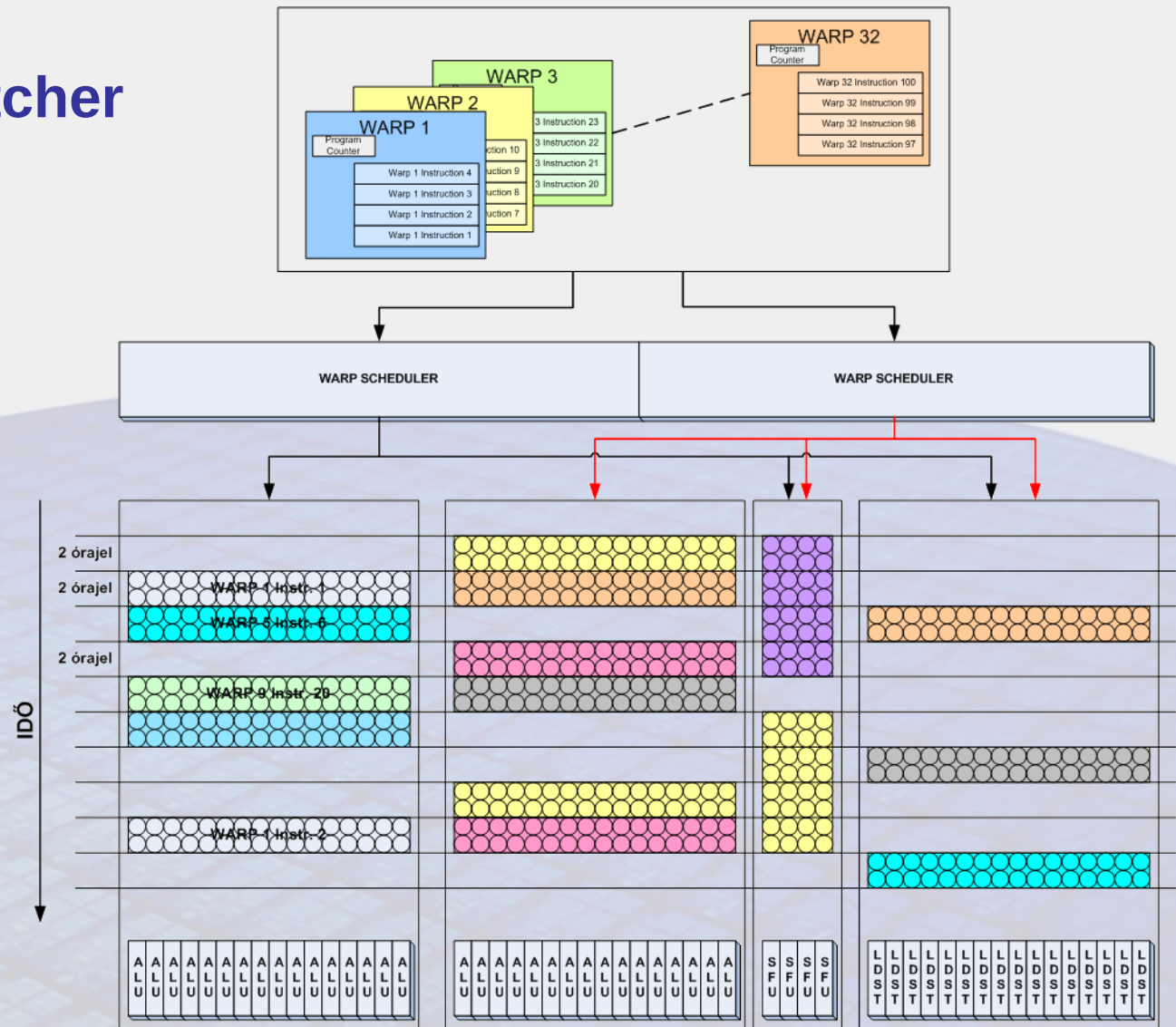
- **Warp: 32 szálból álló csoport**
 - A warp minden szála ugyanazt az utasítást hajtja végre
- **2 Warp ütemező / SM, ALU órajel felén**
 - Mindkettőből 1-1 utasítást dekódol
 - 2*16 utasítás 32 clk alatt
 - 32 ALU
- **Aritmetikai latency ~22 clk**
 - Elrejtés: 22 warp → 704 szál
 - NEM teljesen igaz



GF100 ütemezés

- 2 órajelenként mindkét dispatcher

- 1 ALU vagy
- 1 SFU vagy
- 1 LD/ST



GF104/GF108 – Warp ütemező

- **2 Warp ütemező / SM, ALU órajel felén**
 - 2 Dispatcher/Scheduler → 2-2 utasítást dekódol 1-1 warp-ból
 - $2 \times 2 \times 16$ utasítás 32 clk alatt
 - 48 ALU / SM !!
- **Ha a warp-ból nem lehet két utasítást végrehajtani → csak 32 ALU működik a 48-ból!!!**
- **NEM elég a szál szintű párhuzamosítás (TLP), utasítás szinten is párhuzamosítani kell (ILP)**
 - Összesen 4 utasítás / 2 clk
 - Pontos specifikáció kérdéses (3 ALU + 1 LD/ST ?)

NVIDIA GK110 (Kepler)

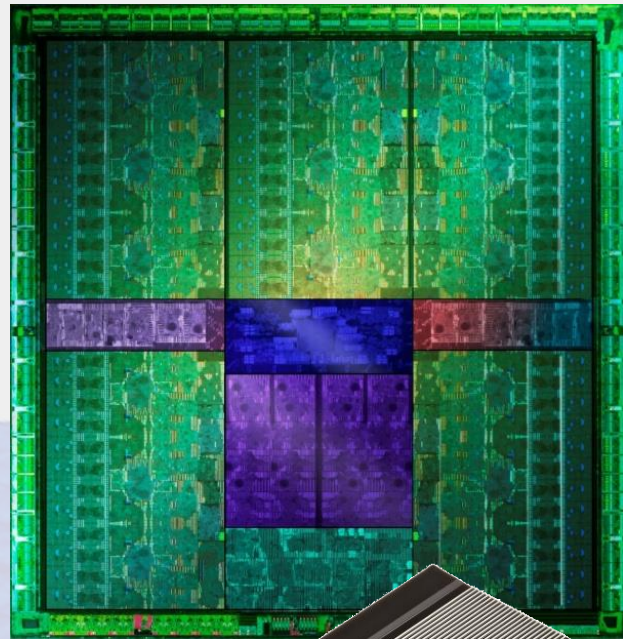
- **NVIDIA GK110B chip (Tesla K40)**

- 2880 ALU
- 7,1 milliárd tranzisztor
- 745 MHz (810/875 Boost)
- 4,29 TFLOPS SP MAD
- 1,43 GFLOPS DP MAD

- **12 GB GDDR5 memória**

- 384 bites busz
- ECC támogatás
- 6000 MHz
- 288 Gbyte/s sávszélesség

- **250 W fogyasztás**



NVIDIA GK110 SMX

- 15 SMX/chip
- 64k x 32bit regiszter
- 192 SP ALU
- 64 DP ALU
- 32 SFU
- 32 LD/ST
- 16 textúrázó egység
- 64 KB Shared Mem.
- 48 KB Data Cache
- A teljes chip egy órajelet használ



GK110 új funkciók (1)

- **Dynamic Parallelism**

- A GPU képes új kernelek indítására, szinkronizációra
- Költséges CPU kernel hívások elhagyása

- **Hyper-Q**

- Ugyanaz a GPU több CPU szálról indított kernel-t képes futtatni
- Max. 32 „work queue” (Fermi: 1)

- **GPUDirect**

- Közvetlen adatátvitel GPU-k között
- Közvetlen adatátvitel GPU és más PCIe eszköz között

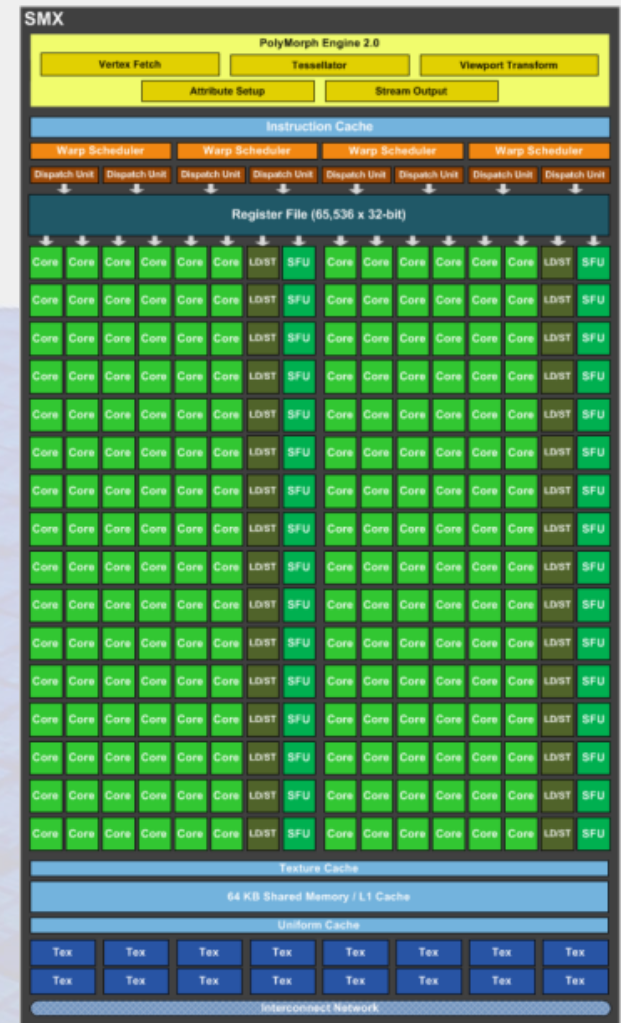
GK110 új funkciók (2)

- **Max 255 regiszter/szál (Fermi: 63)**
- **Shared Memory/L1 Cache**
 - Az eddigi 16 KB/48 KB felosztás mellett lehetséges a 32 KB/32 KB is
 - 64 bites vagy nagyobb load utasítások esetén kétszeres sávszélesség
- **Data Cache**
 - 48 KB, csak olvasható
 - Az eddigi textúra cache, de nem csak textúra olvasó utasításokkal érhető el
 - Fordító automatikusan használja, ha a bemenet konstans

```
const float* __restrict input
```

NVIDIA GK104/GK106

- Kisebb Kepler GPU-k-ban más a lebegőpontos feldolgozóegységek aránya
 - A 192 SP ALU mellett csak 8 DP ALU van
- LD/ST egységek, SFU-k, textúrázók száma megegyezik (32/32/16)
- Nincs Read-Only Cache

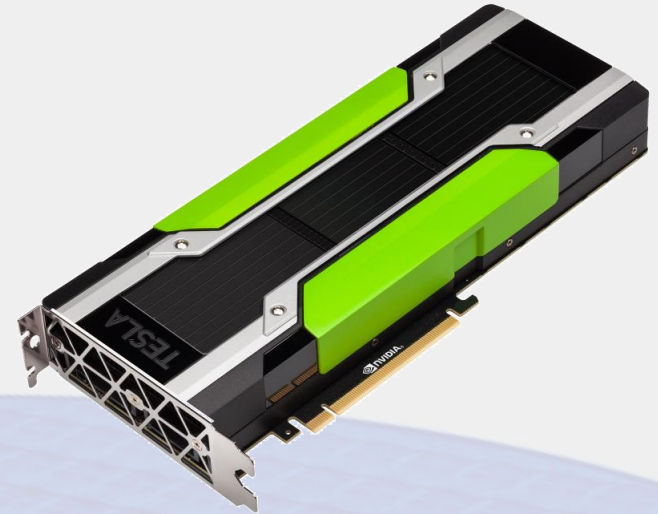


GK110 ütemezés

- **4 darab Warp Scheduler, 8 Instruction Dispatch**
 - 4 Warp ütemezhető egyszerre
 - Minden Warp-ból 2-2 független utasítás
 - DP utasítások párhuzamosan ütemezhetők más utasításokkal
- **Ha nincs független utasítás a Warp-okban**
 - 4 Warp $\rightarrow 4 \cdot 32 = 128$ utasítás ütemezhető
 - A teljes kihasználtsághoz utasítás szintű párhuzamosítás szükséges (mint a GF104 esetében)

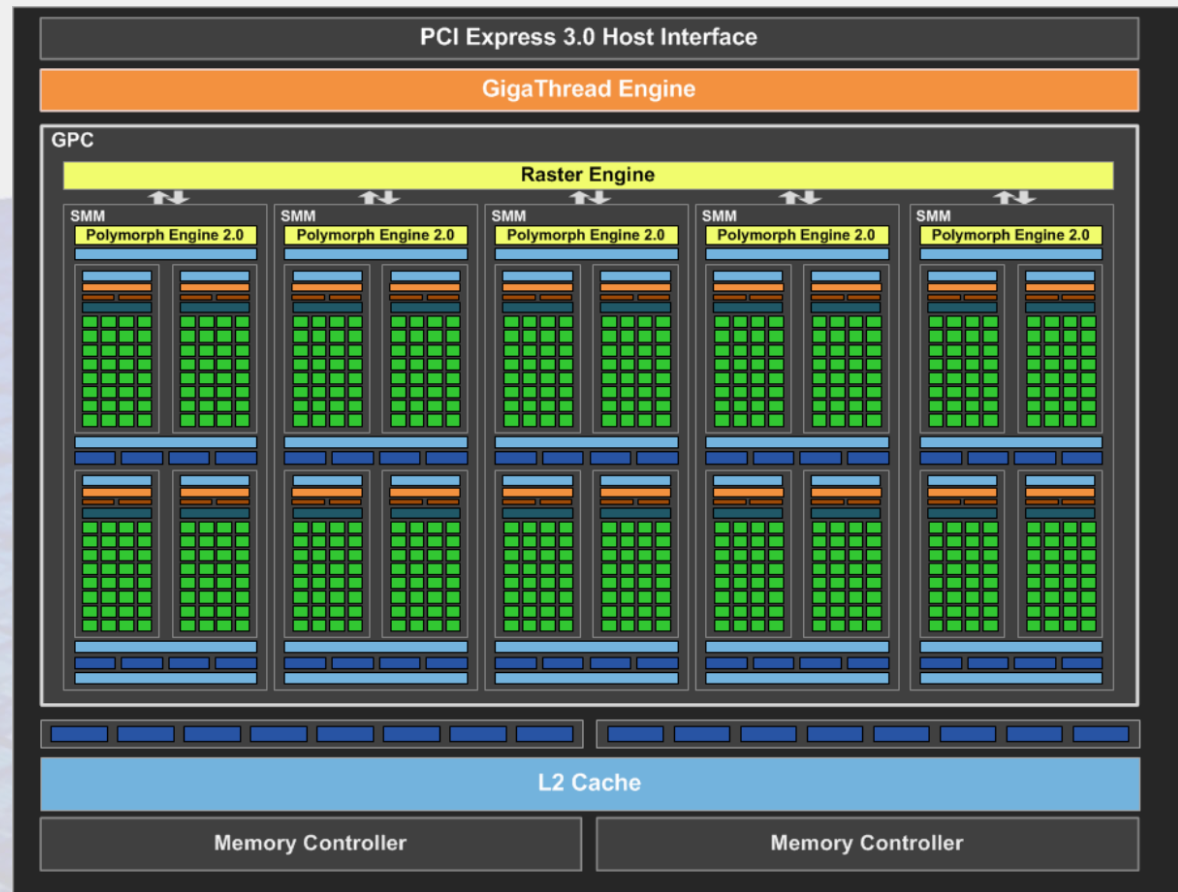
NVIDIA GM100 (Tesla M40)

- **NVIDIA GM104 chip**
 - 3072 ALU, 24 SMM
 - 7,1 milliárd tranzisztor
 - 1140 (Boost) MHz
 - 7 TFLOPS SP MAD
 - 0,21 GFLOPS DP MAD (!!!)
- **12 GB DDR5 memória**
 - 384 bites busz
 - ECC támogatás
 - 600 MHz
 - 288 Gbyte/s sávszélesség
- **250 W fogyasztás**



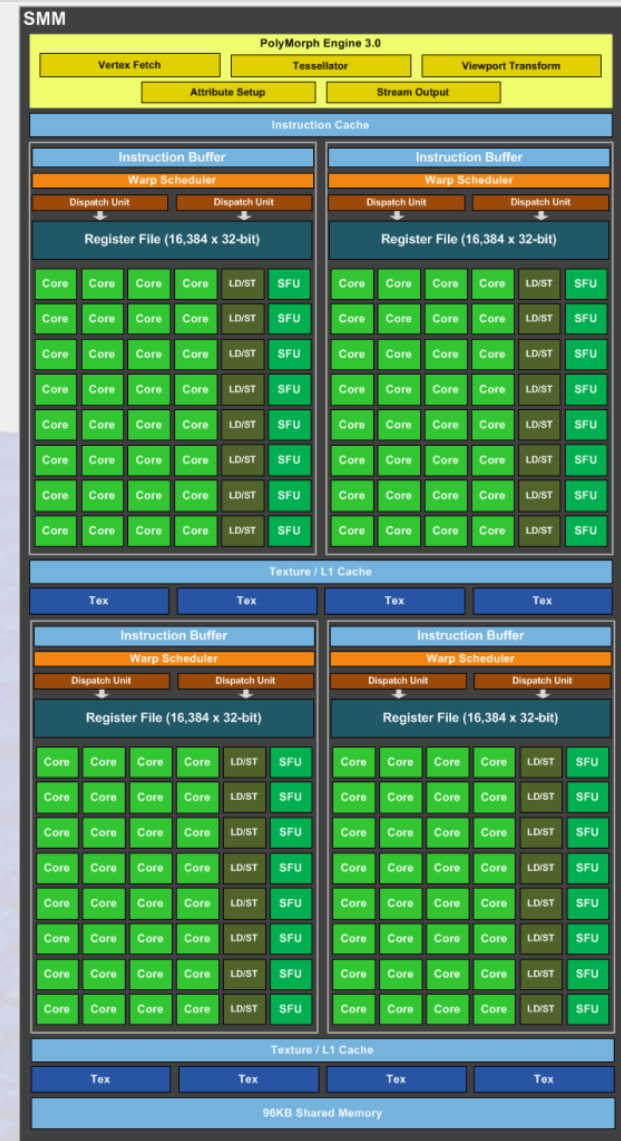
NVIDIA Maxwell (GM107)

- Az új architektúra első terméke – elsősorban középkategóriás játékos GPU
- Cél: nagyon jó teljesítmény/fogyasztás mutató
 - Azonos gyártás-technológián Kepler 2x-esé



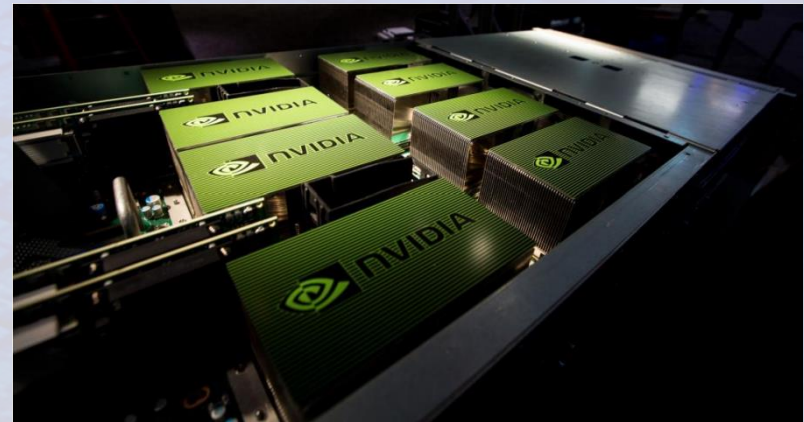
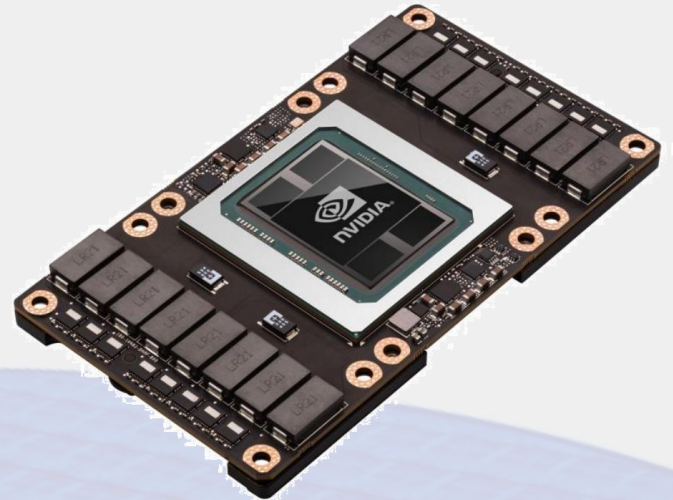
NVIDIA Maxwell v2 SMM

- Maxwell SM = SMM
 - 4x32 ALU / SMM
 - 64k 32 bit regiszter / SMM
 - 4 warp ütemező, nincs megosztva az ALU-k között
 - Csökkentett pipeline latency
 - L1 cache és textúra cache összevonva (48 kB / SM)
 - Shared Memory önállóan
 - 96 KB Shared Memory / SMM
 - Max. 48 KB / thread block
 - L2 cache: 2 MB



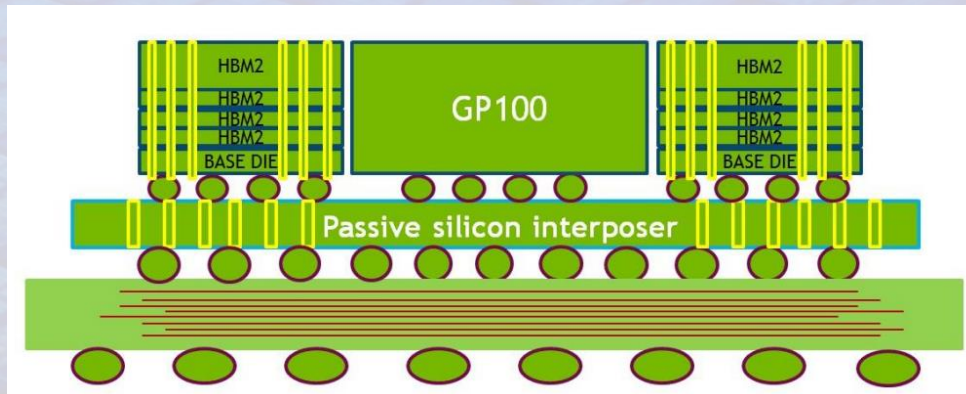
NVIDIA GP100

- **NVIDIA GP100 chip (P100 accelerator)**
 - 3584 ALU
 - 1328 MHz (1480 Boost)
 - 10,6 TFLOPS SP MAD
 - 5,3 TFLOPS DP MAD
 - 21,2 TFLOPS HP MAD
 - NVLINK interface
- **16 GB HBM2 memória (chip-on-wafer-on-substrate)**
 - 4096 bites busz
 - ECC támogatás
 - 1406 MHz átviteli sebesség
 - 720 Gbyte/s sávszélesség (!!!)
- **300 W fogyasztás**



GP100 újdonságok

- **NVLINK (PCIe mellett, nem helyett)**
 - GPU – GPU összeköttetés
 - Egyes IBM Power8/9 CPU-kban is megtalálható
 - 160 GB/s, kétirányú
- **Unified Memory – egységes (virtuális) memória címtér**
 - 49 bit → 512 TB
- **Compute Preemption**
 - Hosszú ideig futó alkalmazás mellé más is ütemezhető
- **HBM2 memória**



GP100

- 60 SM; 2 SM/TPC (Texture Processing Cluster)
- 64 SP ALU/SM, 32 DP ALU/SM
- 8 Memória vezérlő, 512 KB cache/vezérlő (Σ 4 MB)



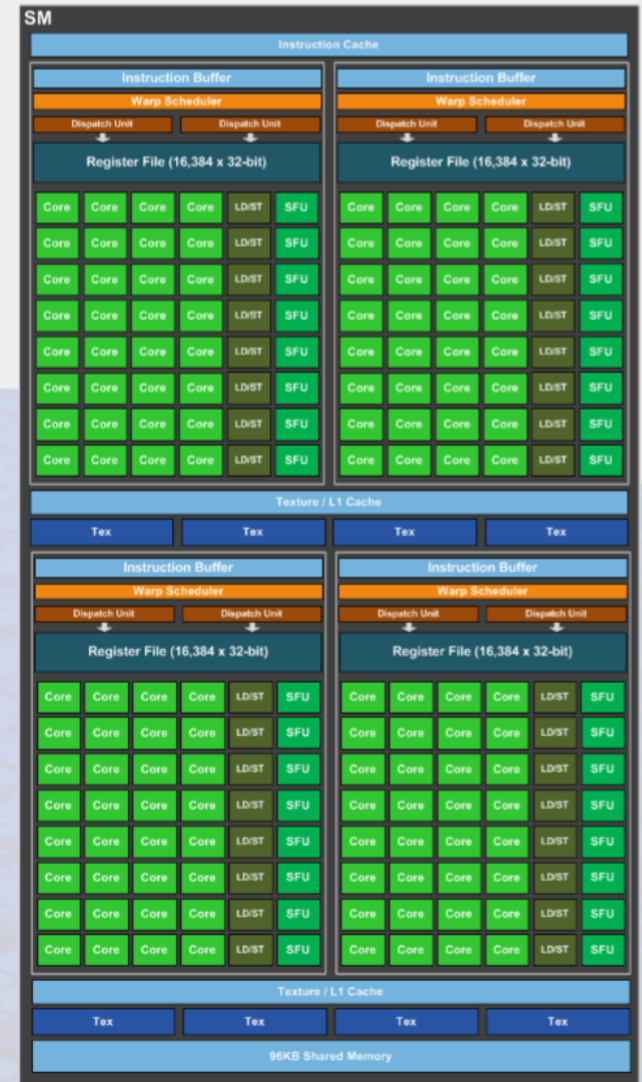
GP100 SM

- FP32/FP64 ALU szétválasztva, FP16 nem
- 64 KB SMEM / SM
 - Egy Thread Block max. 32 KB SMEM-t tud használni
- 24 KB L1 cache / SM



GP104 SM

- Az általánosabban elérhető Pascal architektúra (GTX 1070/1080)
- Jobban hasonlít a Maxwell-re, mint a GP100-ra
 - Fele akkora regiszter file
 - 64k helyett 96k SMEM, de 64 helyett 128 ALU osztozik rajta



NVIDIA V100

- **NVIDIA Volta GPU**
 - 5120 FP32 ALU (15 Tflops/s)
 - 2560 FP64 ALU (7.5 Tflops/s)
 - 640 Tensor Core (120 Tops/s)
- **16 GB HBM2 memória**
 - 900 GB/s
- **NVLINK 2.0 (vagy PCIe 3.0)**
 - 6x 25 Gbit/s



V100 Streaming Multiprocessor

- Külön INT, FP32 és FP64 ALUk

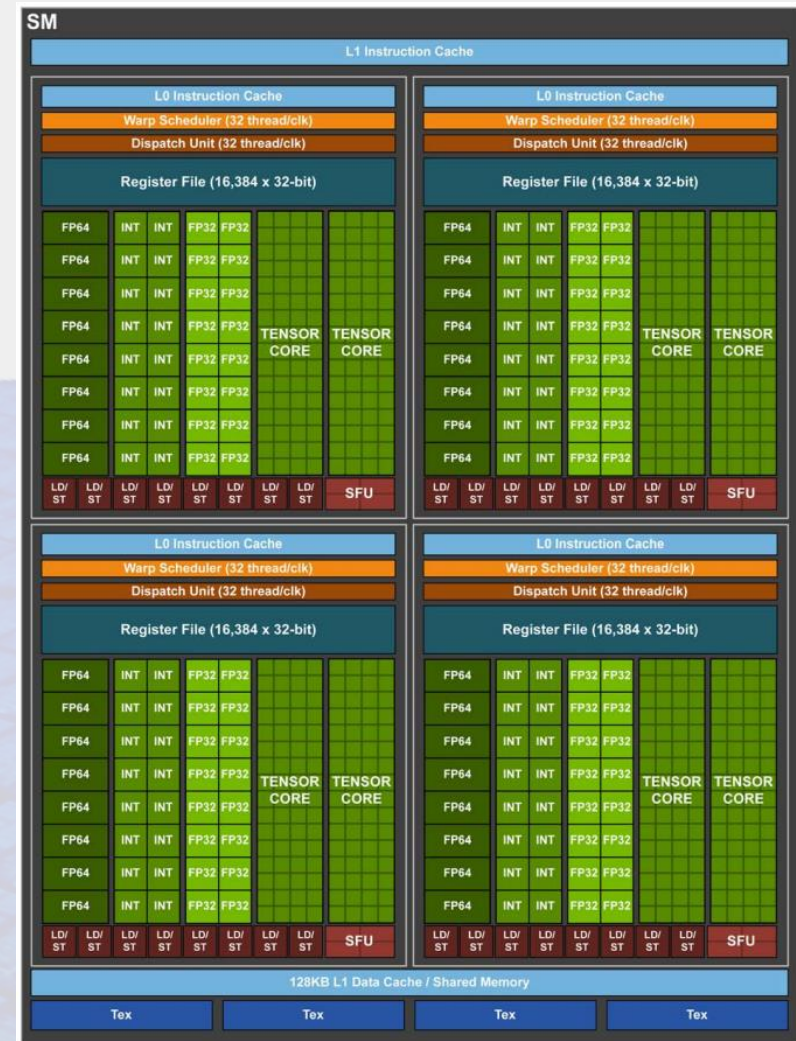
- 64 FP32 / SM
- Konkurens INT és FP

- 8 Tensor Core

- 4x4 mátrix MAD

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

- 1 warp/16 core 2 órajel alatt
- Minden szálhoz külön PC, stack
- 64kx32b regiszter file (== Pascal)
- 128 (96) kB Shared Memory
 - 2x Pascal
 - Egyben adat és textúra cache
- Cache hatékonyság sokkal jobb



NVIDIA TU102

- **NVIDIA Turing architektúra (Quadro RTX 6000)**
 - 4608 FP32 ALU (16,3 Tflops/s)
 - 144 FP64 ALU (0,5 Tflops/s)
 - 576 Tensor Core (130/261/522 Tops/s)
 - 72 Ray Tracing core
 - 1770 MHz órajel
- **24 GB GDDR6 memória**
 - 672 GB/s
- **NVLINK 2.0 x8, PCIe 3.0**



TU102 Streaming Multiprocessor

■ NVIDIA Turing GPU

- 4 “processing block”
- 64 FP, 64 INT ALU konkurens működéssel!
- Tensor Core
 - INT8 és INT4 támogatás 2x és 4x sebességgel (+elvileg INT1)
- 64kx32b regiszter file
- 96 kB egyesített L1 cache, textúra cache és Shared Memory 2x sávszélesség (64+32/32+64)
- 6 MB L2 cache
 - ~2x sávszélesség



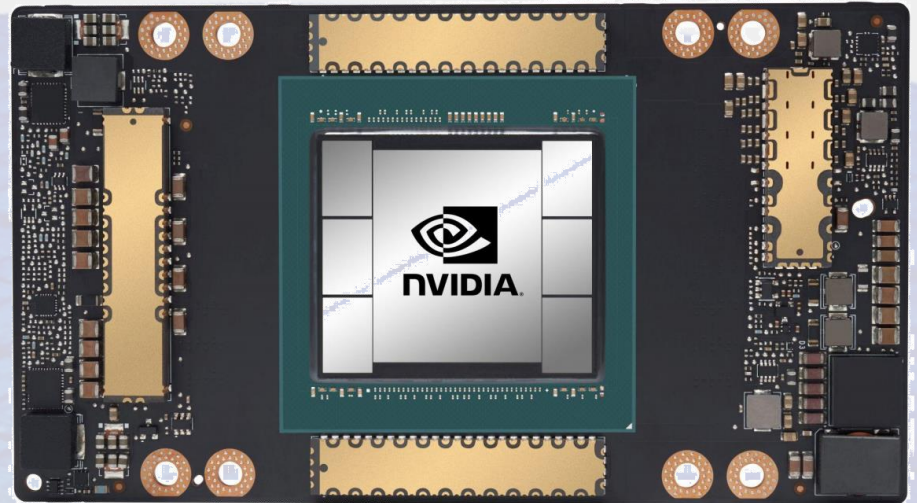
Turing részletek

- **Processing block**
 - 1-1 warp egy block-on fut
 - $\text{Block} = \text{warp_id} \% 4$
 - Legalább $\text{SM} * 128$ szál kell
- **Regiszter fájl: 2 bank, 2x32 bit port**
 - Gond: ha egy bankhoz kettőnél több hozzáférés van egy utasításban
- **Uniform datapath: integer utasítások, saját regiszterekkel**



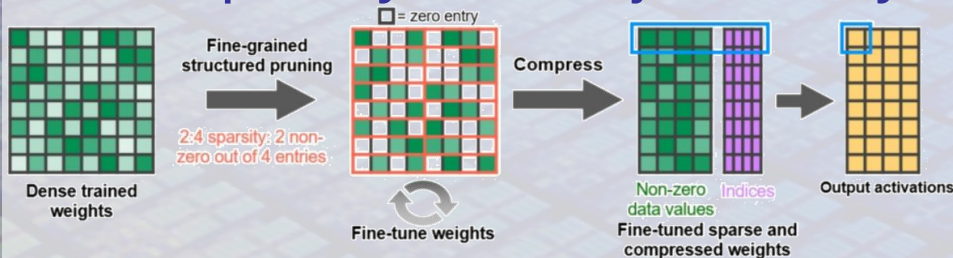
Ampere A100

- **Ampere HPC verziója**
 - ~1400 MHz
 - 6912 FP32 mag – 19,5 Tflops/s
 - Tensor Cores 3.0 – 312 Tflops/s FP16
 - “Multi-Instance GPU”: GPU particionálás felhasználók között
- **40GB HBM2 memória**
 - 1555 GB/s
- **NVLink 3.0**
 - 50 Gb/s
- **PCIe 4.0**
 - 32 Gb/s
 - Single-root I/O virtualization (SR-IOV): fizikai és virtuális PCIe eszközök



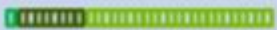
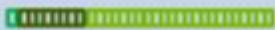
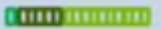
Ampere A100 SM

- Konkurens INT és FP32
- Aszinkron memória másolás (Global → Shared)
- Warp-szintű redukció instr.
- 192 kB Shared Memory/L1
- Tensor Cores
 - 256 FP16 FMA/core
 - FP16, BF16, TF32, FP64, INT8, INT4, bináris
 - Sparsity – 2x teljesítmény



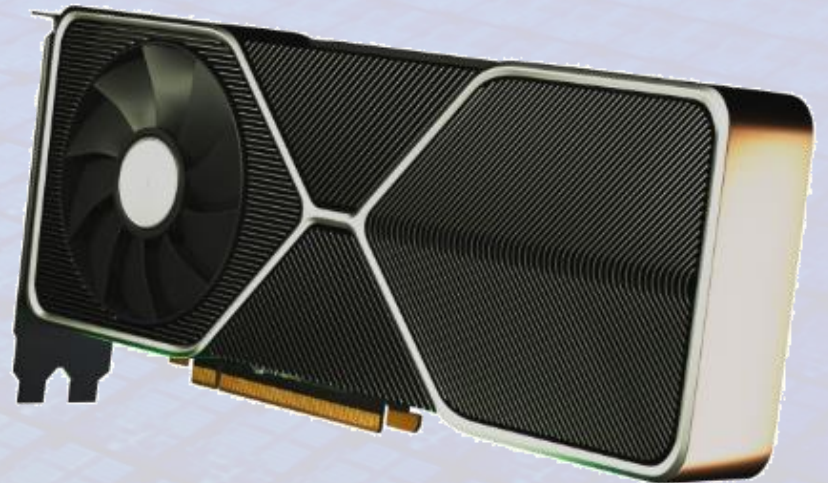
Tensor core formátumok

- FP32: 1 bit előjel, 8 bit exponens, 23 bit mantissza
- TF32: 1 bit előjel, 8 bit exponens, 10 bit mantissza
- FP16: 1 bit előjel, 5 bit exponens, 10 bit mantissza
- BF16: 1 bit előjel, 8 bit exponens, 7 bit mantissza

	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA
A100	FP32 	FP32 	19.5	1x
	TF32 	FP32 	156	8x
	FP16 	FP32 	312	16x
	BF16 	FP32 	312	16x
	FP16 	FP16 	312	16x
	INT8 	INT32 	624	32x
	INT4 	INT32 	1248	64x
	BINARY 	INT32 	4992	256x
	IEEE FP64 		19.5	1x

Ampere A102

- **GeForce RTX 3080**
 - ~1,7 GHz
 - 8704 FP32 mag – 29,8 Tflops/s
 - 272 v3 Tensor Core – 119 Tflops/s FP16
- **10 GB GDDR6X memória**
 - 320 bit interfész @ 19Gbps, 760 GB/s
- **NVLink 3.0**
- **PCIe 4.0**



Ampere A102 SM

- INT/FP32 + INT
- 2 FP64 mag / SM
- Tensor Core 3.0
 - 128 FP16 FMA / Core
 - Sparsity
- 128 kB Shared Memory/L1
- 2. gen. RT Core



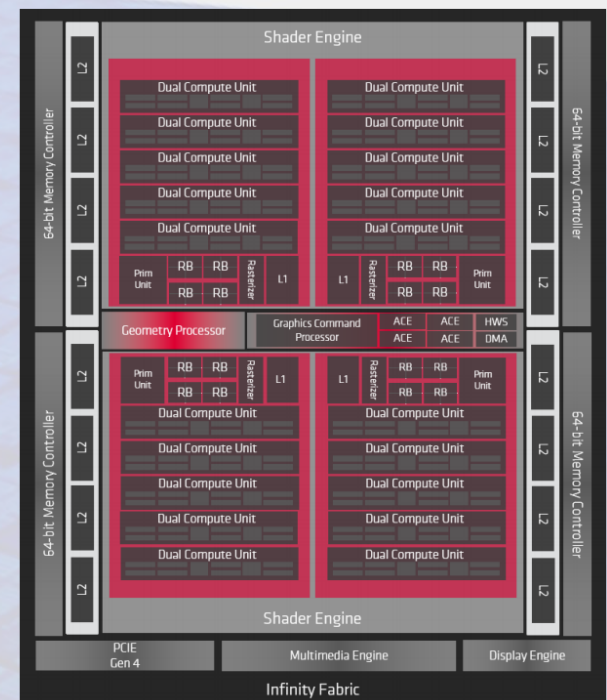
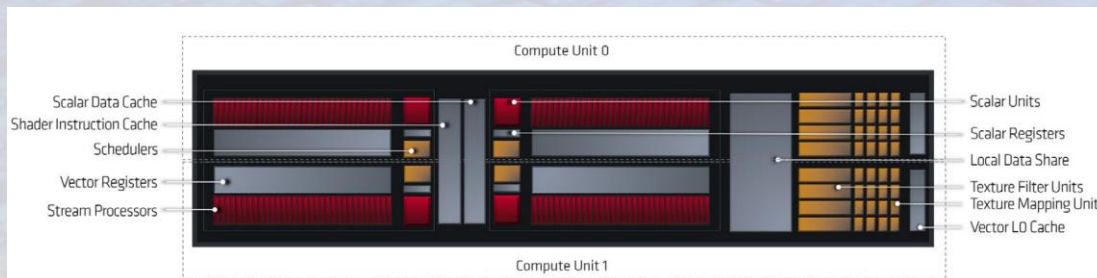
Teljesítmény

■ Órajelenként végrehajtott utasítások száma, SM-enként

	2.0 (Fermi)	2.1 (Fermi)	3.5 (Kepler)	5.0/5.2 (Maxwell)	6.0 (Pascal GP100)	7.X (Volta, Turing)
ALU/SM	32	48	192	128	64	64
FP ADD, MUL, MADD	32	48	192	128	64 FP16: 128	64 FP16:128
FP SFU (sqr, log, exp)	4	8	32	32	16	16
INT ADD, SUB	32	48	160	128	64	64
INT MUL, MADD	16	16	32	Több	Több	64
SHIFT	16	16	32/64	64	32	64
CMPR, MIN, MAX	32	48	160	64	32	64
Logikai műveletek	32	48	160	128	64	64
Lead 0, MSB, POPCNT	16	16	32	Több/32	Több/16	16
Típus konv. 32 bites típusra	16	16	128	32	16	16

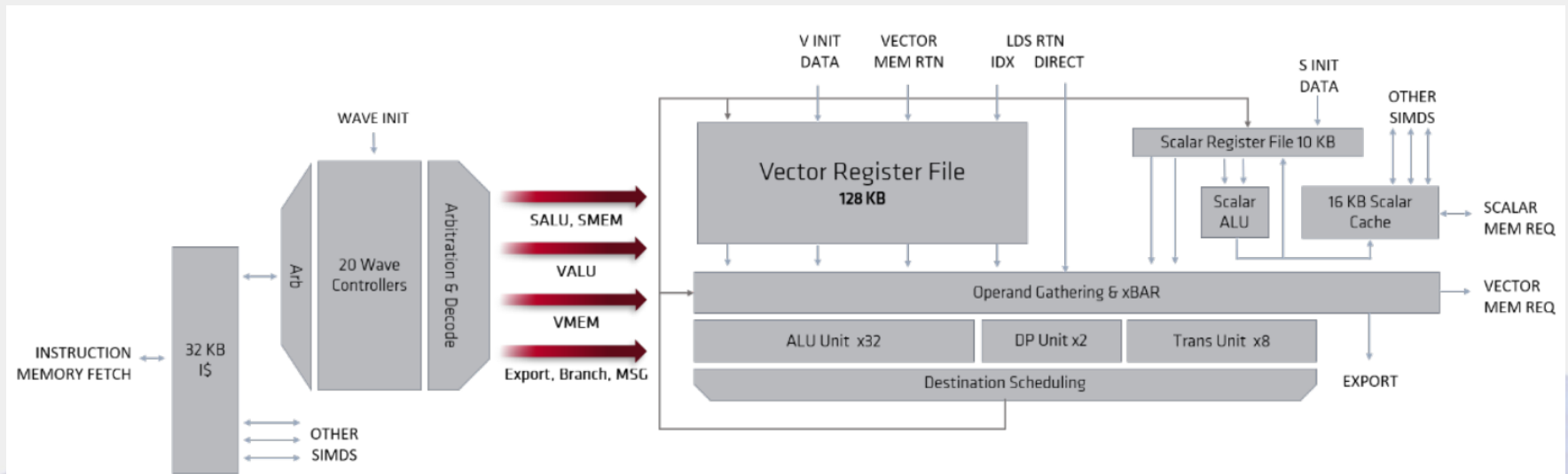
AMD (1)

- Unified Shader ~2008 környékén
- Múlt
 - TeraScale architektúra: VLIW5, ill. VLIW4 utasításkészlet
 - GCN architektúra: skalár + vector, már nem VLIW
- Jelen
 - RDNA: GCN továbbfejlesztése
 - HPC termékek még GCN (Vega) architektúrára épülnek

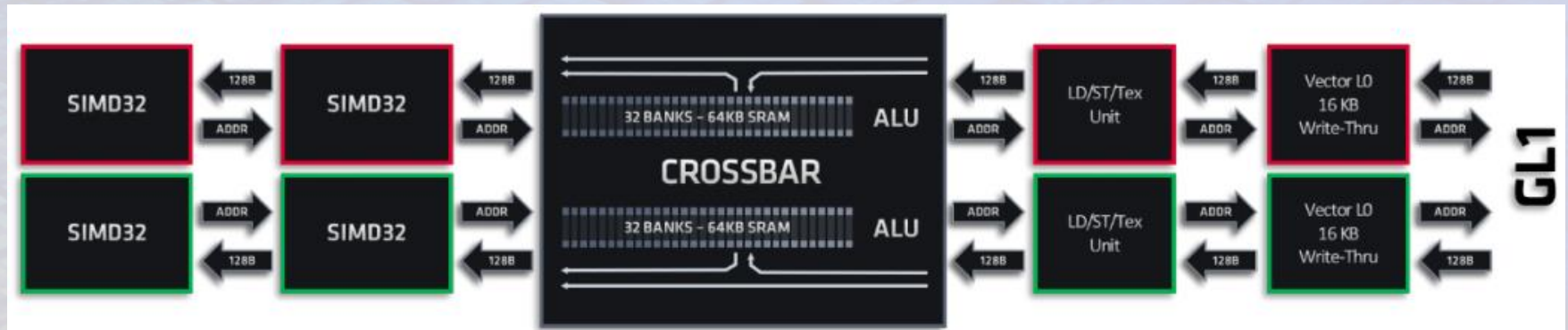


AMD (2)

■ RDNA Compute Unit

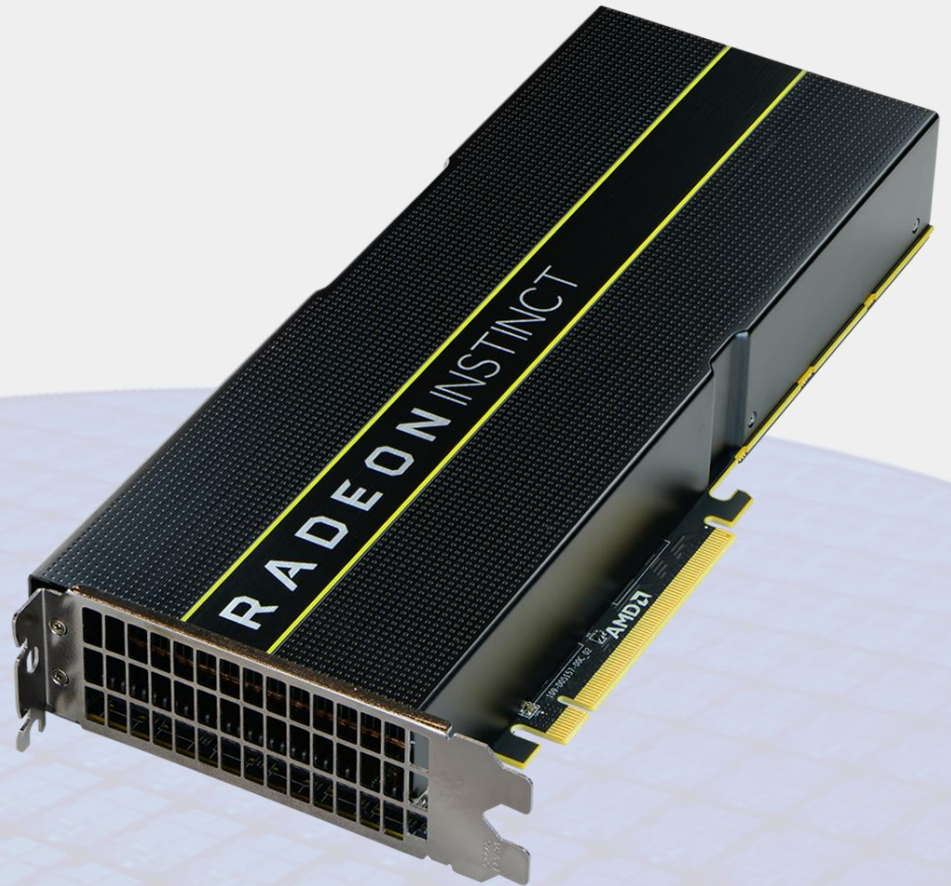


■ RDNA memória elérés



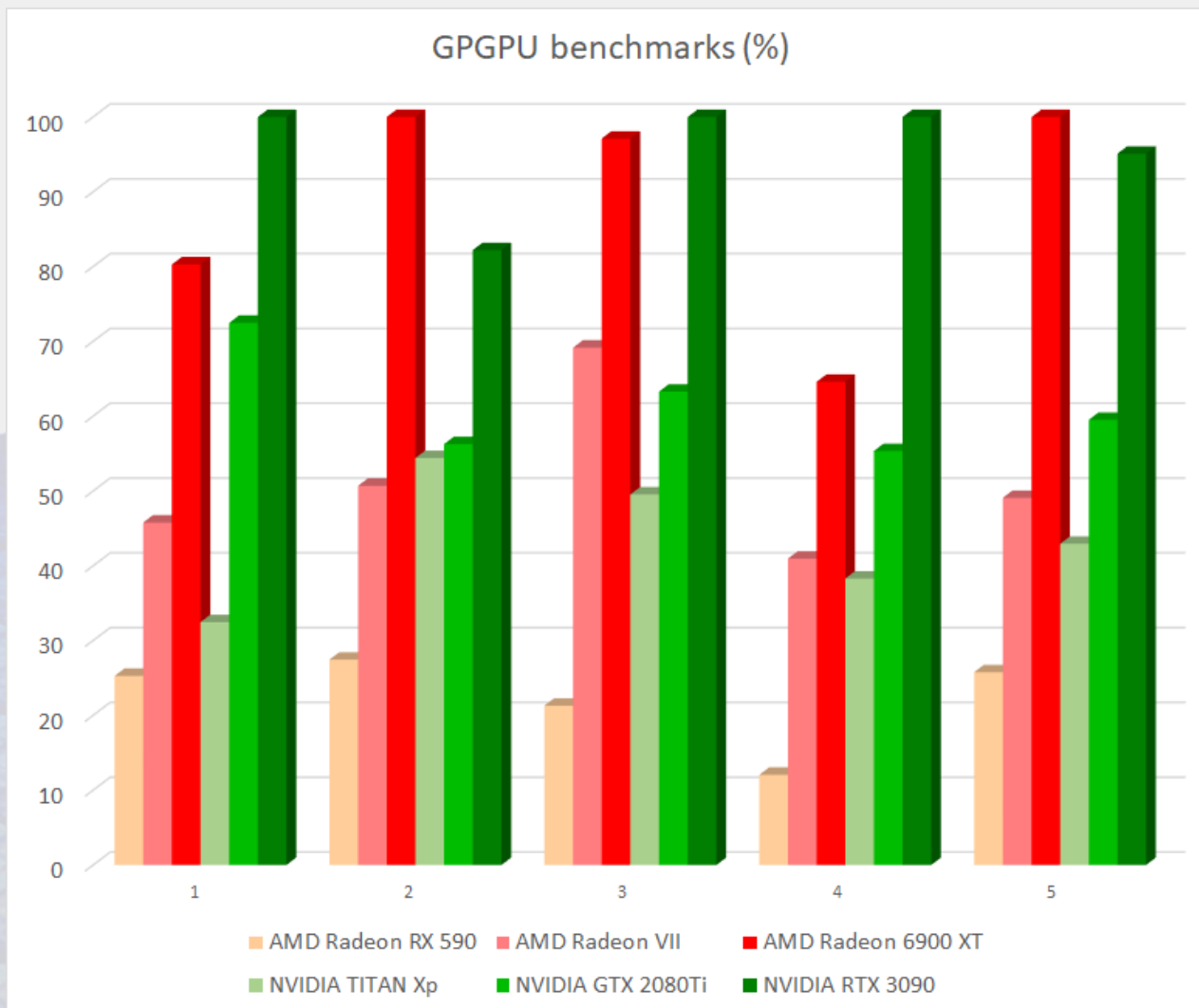
AMD Radeon Instinct MI25

- AMD Vega architektúra
- 4096 Stream Processor
 - 12,3 Tflops/s FP32
 - 0,768 Tflops/s FP64
 - 24,6 Tflops/s FP16
- 16 GB HBM2 memória
 - 484 GB/s
- PCIe 3.0 x16
- 300W



GPU-k relatív teljesítménye

- A GPU-k relatív teljesítménye eléggé alkalmazás függő, de....



SM vs. SMX vs. CU

	GF100	GK110	GM100	GP100	GCN	egység
ALU	32	192 SP 64 DP	4x32	2x32	64	
Max. szál	1536	2048	2048	2048	2560	
Regiszter	32k	64k	4x16k	2x32k	64k	32 bit szó
Shared Memory	64 16 – 48 felosztás	64 16 – 32 – 48 felosztás	96	64	64	kbyte
Cache			48	24	16	kbyte
Read-Only Cache	-	48	-	-	-	kbyte
Max. regiszter / szál	63	255	255	255	128	
SP FLOP / clk	64	384	4x64	64	128	
SM BW / clk	64	128 (256)	128	256	128	byte
Cache BW / clk			?	?	64	byte
Regiszter / ALU	1024	341,3	512	1024	1024	szó
Sh. Mem / ALU	1,5	0,25	0,25	1	1	kbyte
L1 Cache / ALU			0,375	0,375	0,25	kbyte
SM BW / FLOP	1	0,33 (0,66)	0,5	2	1	byte/flop
Cache BW / FLOP			?	?	0,5	byte/flop

CUDA nyelvi elemek (1)

- **__host__ int func(...);**
 - Hoszton futó és onnan hívható függvény
- **__global__ void kernel(...);**
 - hoszt által hívható kernel függvény
- **__device__ void kernel(...);**
 - kernelből hívható kernel függvény
- **__device__ int var;**
 - globális (külső) memóriában deklarált változó
- **__shared__ int var;**
 - Shared Memory-ban tárolt változó
- **__constant__ int var;**
 - konstans memóriában tárolt változó

CUDA nyelvi elemek (2)

- **Kitüntetett kernel változók**
 - threadIdx, blockIdx, blockDim
- **Kernel változó típusok**
 - 2, 3, 4 elemű vektor változók
 - pl. int4, float4,
 - elemei: x, y ,z, w
 - konstruktor: int2 make_int2(int x, int y);
- **Matematikai függvények**
 - sin, cos, tan, log, exp, pow
- **Textúra kezelő függvények**

CUDA → OpenCL

▪ Kernel kulcsszavak

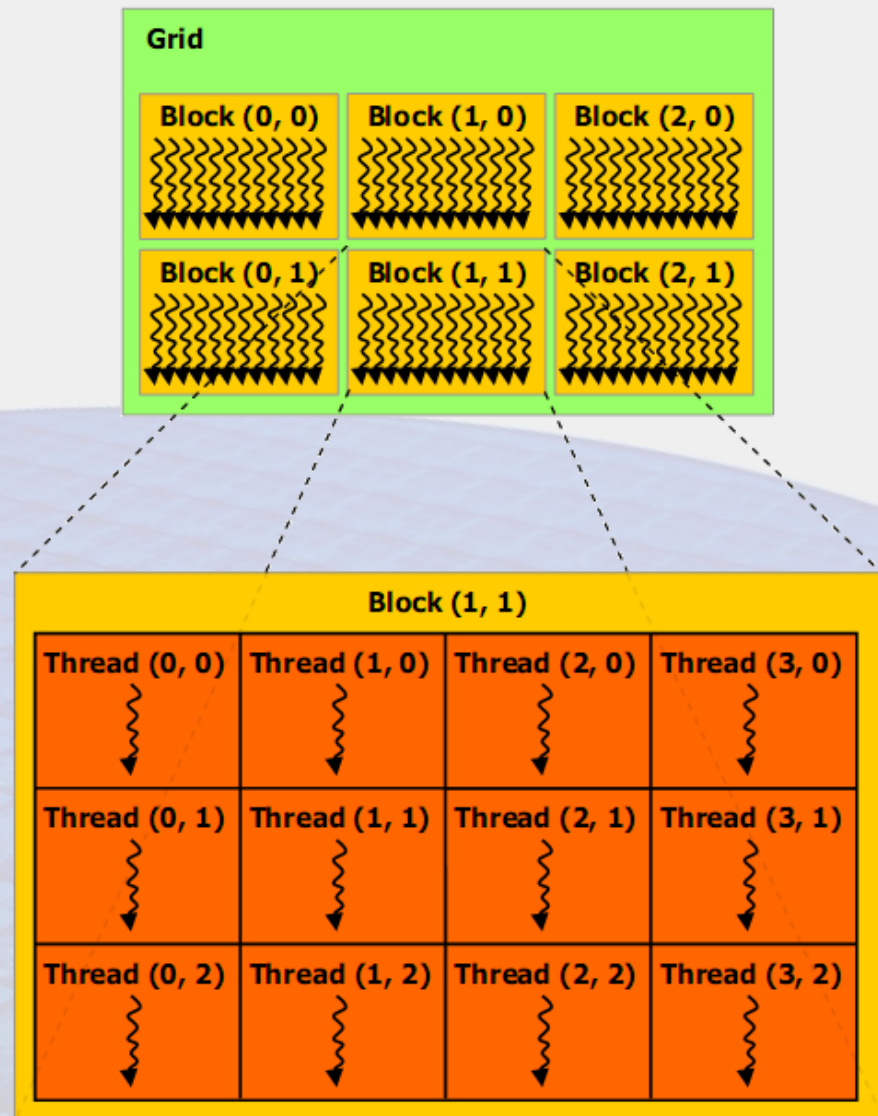
CUDA	OpenCL
__global__	__kernel
__device__	---
__constant__	__constant
__device__	__global
__shared__	__local

▪ Szál azonosítók

CUDA	OpenCL
threadIdx.x	get_local_id(0)
blockIdx.x	get_group_id(0)
blockDim.x	get_local_size(0)
	get_global_id()

CUDA – szál hierarchia

- **Kernel:** C kód, minden szál ezt futtatja
- **Thread block:** egy SM-en futó szálak csoportja
 - legfeljebb 3D:
 $\{x,y,z\} = \{512,512,64\}$
 - Warp: egyszerre futó szálak → !divergens elágazások
- **Grid:** Thread block-ok mátrixa
 - 3D:
 $\{x,y\} = \{65535, 65536\}$



Szálak - kommunikáció

- **Thread Block-on belül**
 - Explicit szinkronizáció
 - Atomikus műveletek
 - Adat megosztás
- **Thread Block-ok között**
 - NINCS explicit szinkronizáció
 - Adat megosztás és atomi műveletek a globális memóriában → LASSÚ
 - (Architektúra függően L2 cache-ben, az gyorsabb)

Thread Block

- **Thread Block – SM (Streaming Multiprocessor)**
 - Egy Thread Block egy SM-en fut
 - Az SM-en időosztásban több TB is futhat
 - Latency elrejtés!
 - Aritmetikai: ~22 órajel
 - Memória: ~400 órajel
 - Limitált erőforrások
 - Regiszter file mérete (32-bit)
 - 32k – 64k (GPU függő)
 - Shared Memory mérete
 - 16k – 64k (GPU és beállítás függő)

Thread Block – SM

- **Regiszter file méret**
 - Egy szál 24 regisztert igényel
 - 256 szál / Thread Block
 - $32768 / (256 * 24) \rightarrow 5$ TB fér el a regiszter file-ban
- **Shared memory**
 - 12.000 byte / Thread Block
 - $49152 / 12000 = 4 \rightarrow 4\text{TB}$ fér el a Sh. Mem.-ben
- **Ütemezett szálak**
 - 48 warp / SM; 1536 szál / SM
 - $1536 / 256 = 6 \rightarrow 6$ TB / SM
- **Konklúzió: 4 TB / SM $\rightarrow 4 * 256 = 1024$ szál**
 - $1024 / 1536 = 66\% \rightarrow$ Occupancy

CUDA: memória típusok

- **Shared Memory – 16/32 kbyte / SM**
 - 16/32 bank, warp ütközésmentesen elérheti
 - Minden i. cím az i. bank-hoz tartozik
 - Broadcast olvasás
 - Atomi operáció
- **Constant Memory – 64 kB**
 - Csak olvasható, cache-elt (8kB / SM) külső memória
- **Texture Memory**
 - Csak olvasható, 2D/3D lokális hozzáférésre optimalizált cache-elt külső memória (cache: 8 kB / SM)
- **Global Memory**
 - Írható/olvasható külső memória Fermi óta cache-elt
 - Van atomi operáció (összes szádra)
- **Local Memory**
 - Thread-ekhez allokált külső memória; Fermi óta cache-elt

CUDA occupancy calculator

■ CSAK iránymutató

- Nagyobb foglaltság != gyorsabb végrehajtás !!!

CUDA GPU Occupancy Calculator

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 2.1 (Help)
1.b) Select Shared Memory Size Config (bytes) 49152

2.) Enter your resource usage:
Threads Per Block 256 (Help)
Registers Per Thread 24
Shared Memory Per Block (bytes) 12000

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
Active Threads per Multiprocessor 1024 (Help)
Active Warps per Multiprocessor 32
Active Thread Blocks per Multiprocessor 4
Occupancy of each Multiprocessor 67%

Physical Limits for GPU Compute Capability: 2.1

Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity (for register allocation)	0
Maximum Thread Block Size	1024

Allocation Per Thread Block

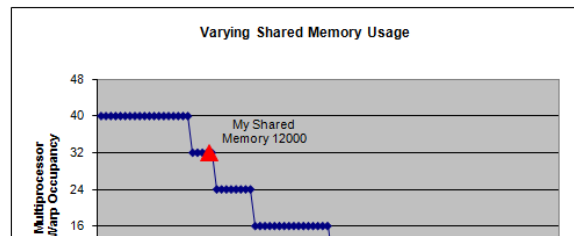
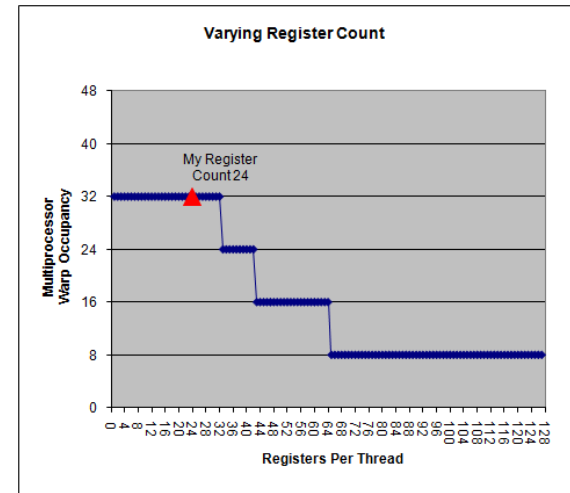
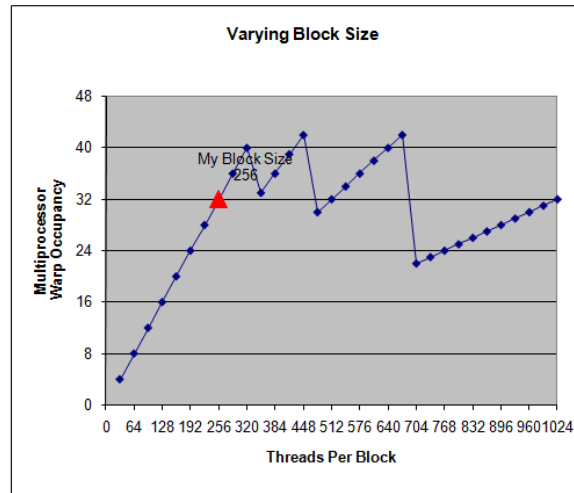
Warps	8
Registers	6144
Shared Memory	12032

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	6
Limited by Registers per Multiprocessor	5

Your chosen resource usage is indicated by the red triangle on the graphs.

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



Hoszt memória foglalás (1.)

- Sztenderd C függvények: `malloc()`, `_aligned_malloc()`
 - Lapozható (pageable) memóriát foglal
- `cudaError_t cudaHostAlloc(void **ptr, size_t size, unsigned int flags)`
 - `cudaHostAllocDefault`
 - Nem lapozható memória
 - Rendszer memória – GPU memória másolás átlapolható a kernel futtatással
 - Sok esetben gyorsabb adatátvitel
 - (Közös CPU- GPU memória esetén nem kell másolás)
 - `cudaHostAllocPortable`
 - Több GPU-s rendszer esetén minden GPU-hoz

Hoszt memória foglalás (2.)

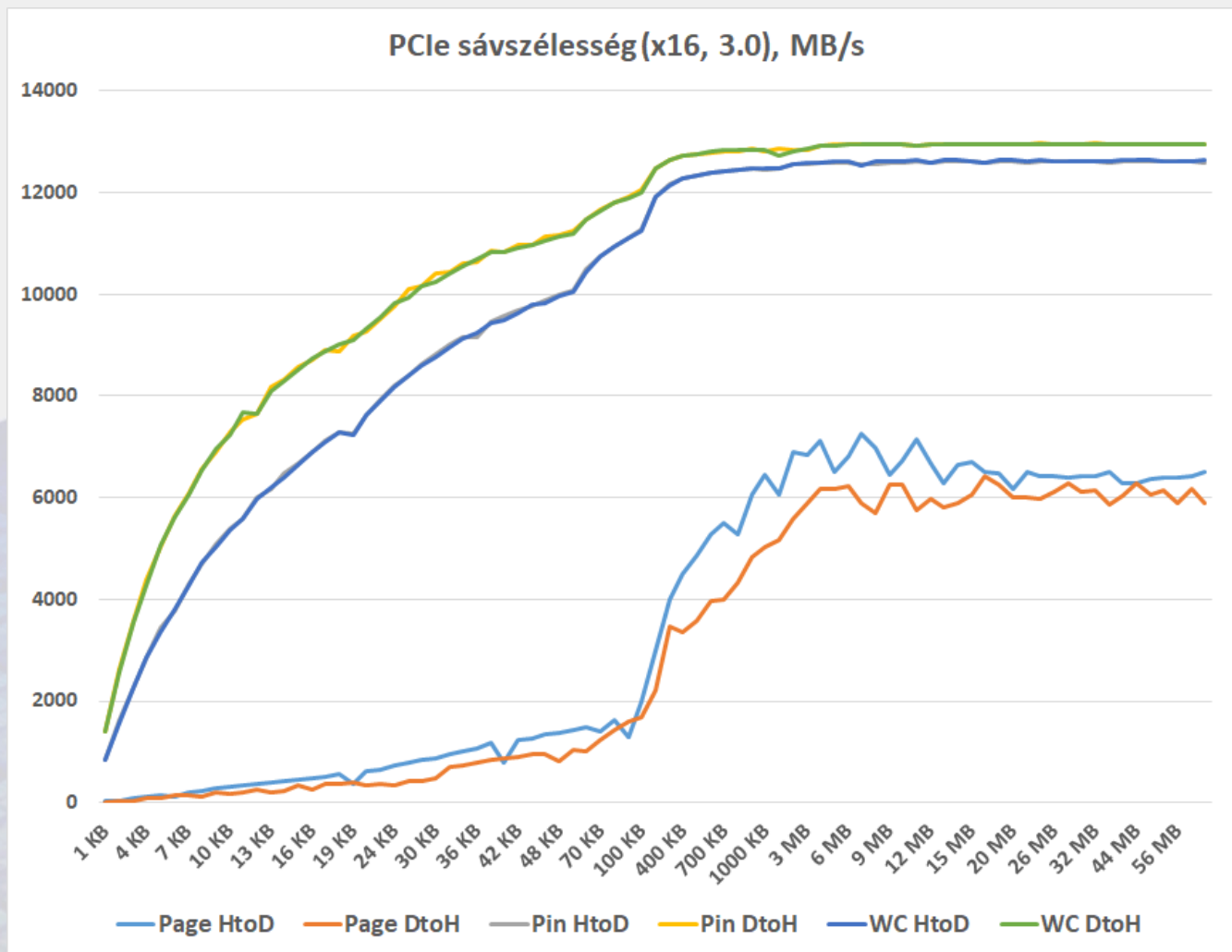
- `cudaError_t cudaHostAlloc(void **ptr, size_t size, unsigned int flags)`
 - `cudaHostAllocWriteCombined`
 - Kikerüli a Cache-t
 - gyors PCIe írás, lassú CPU olvasás
 - `cudaHostAllocMapped`
 - GPU részéről is elérhető: `cudaHostGetDevicePointer()`
 - A hoszt – GPU memória másolás automatikusan megtörténik
 - Stream (lásd később) nélküli adatmásolás – kernel futtatás átlapolás
- **Felszabadítás: `cudaFreeHost()`**

GPU memória foglalás

- **cudaError_t cudaMalloc(void ** devPtr, size_t size)**
 - size byte-nyi GPU memória foglalása
- **cudaError_t cudaFree()**
 - GPU memória felszabadítása
- **cudaError_t cudaMallocArray()**
 - 2D tömb foglalása GPU memóriában (textúra)
- **cudaError_t cudaMallocPitch()**
 - 2D tömb, minden sor megfelelő (pl. 128 byte) határon kezdődik
- **cudaError_t cudaMalloc3D()**
 - 3D tömb megfelelő határra helyezve

PCIe átviteli sebesség

- Elsősorban a chipset-től függ!



Globális memória (1)

- **Cache line**

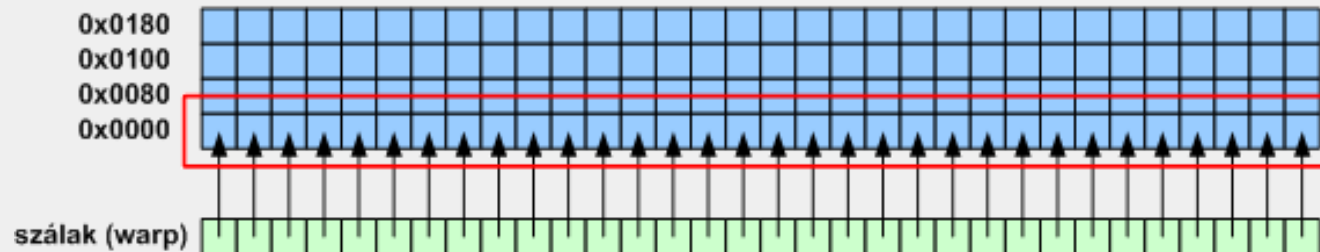
- L1: 128 byte → 128 byte-os memória hozzáférés
- L2: 32 byte → 32 byte-os memória hozzáférés

- **Mi hol cache-elt?**

- CC 2.x: minden hozzáférés L1-ben és L2-ben
- CC 3.x
 - Global memory: L2
 - 3.5, 3.7: lehet RO cache, L1 is
 - Local memory: L1 & L2
- CC 5.x-től: alapvetően CC 3.0
 - Read-only: cache-elhető L1-ben
 - Read/Write: 5.0 csak L2; 5.2-nél lehet L1 is

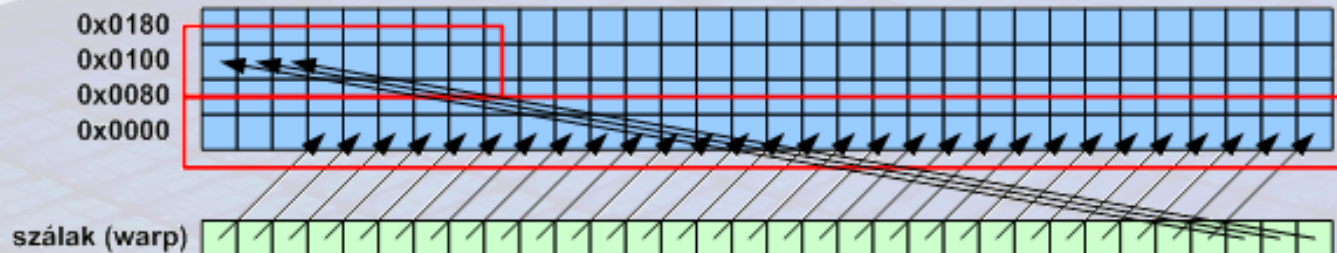
Globális memória (2)

- 128 byte/warp, aligned hozzáférés



- L1: 1x128 byte; L2: 4x32 byte

- 128 byte/warp, unaligned hozzáférés

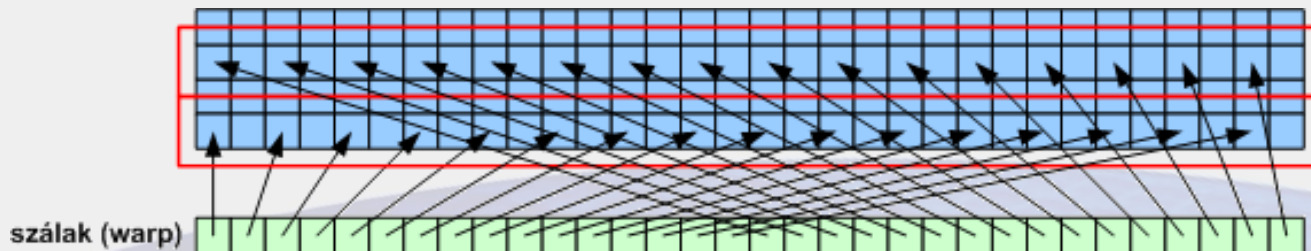


- L1: 2x128 byte; L2: 5x32 byte

Globális memória (3)

- „Stride” hozzáférés

- Minden szál a sorszáma*S címet éri el
- Pl. $S=2$



- L1: 2x128 byte; L2: 8x32 byte → KERÜLENDŐ
- **Törekedni kell az ideális hozzáférésre!!!**
 - 128 byte/warp, 128 byte-os határra illesztve
 - 256 byte/warp és 512 byte/warp is jó lehet
- Több szál írja ugyanazt a címet: egy írás történik, nem definiált, hogy melyik

Lokális & konstans memória

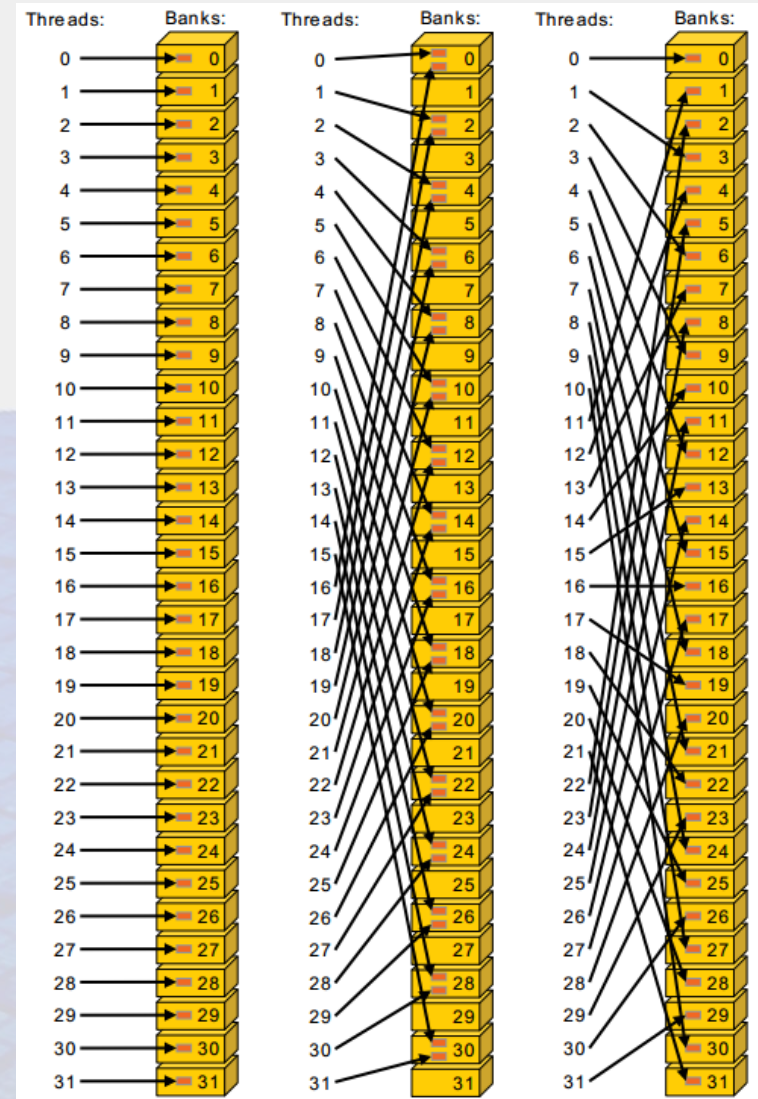
- **Lokális memória: fizikailag külső memóriában**
 - Automatikusan minimális számú memória művelettel jár
 - Fermitől kezdve cache-elt
 - Regiszterek átmeneti tárolása (register spilling) → sokat lassíthat!
 - Nem fordítási időben eldönthető címzésű tömbök!
- **Konstans memória**
 - Külső memória hozzáférés csak cache-miss esetén van
 - Optimális esetben a teljes warp ugyanazt a címet olvassa
 - Különböző címről olvasások esetén lineárisan csökken a sebesség

Shared memory (1.)

- **32 bank a warp 32 szálához**
 - Az egymást követő memória címek más-más bank-hoz tartoznak
 - Vektor változók (és double) több bankot foglalnak!
- **Optimális:**
 - Minden szál ugyanazt a címet olvassa (broadcast)
 - Minden szál más bankot olvas:
 - Cím: + threadIdx.x
- **Bank ütközés esetén a szálak szekvenciálisan hajtódnak végre!**

Shared Memory (2.)

- 1: A Warp minden szála egymás utáni bankot olvas (i-edik szál i-edik bankot) → nincs bank ütközés
- 2: Az i-edik szál a $2 \cdot i$ -edik bankot olvassa → 2-szeres bank ütközés
- 3: Az i-edik szál a $3 \cdot i$ -edik bankot olvassa → nincs bank ütközés
- Akkor sincs, ha teljesen random a szál – bank összerendelés, de nincs átfedés



Textúrák

- **Textúra memória csak olvasható, cache-elt**
 - Fermi-től kezdve írható, de nem koherens (surface)
 - Fermi: globális memória cache gyorsabb lehet
- **Textúra: a textúra memória egy darabja**
 - Egy eleme: texel
 - Hivatkozás: textúra referencia, ezek átlapolódhatnak
 - 1, 2 vagy 3 dimenziós
 - Egy texel 1, 2 vagy 4 komponensből állhat
 - 8, 16, 32 bit INT, 32 bit FP (driver API: 16 bit FP)
- **Textúra referencia**
 - Számos paramétere fordítási időben dől el

Textúra referencia - deklaráció

```
texture<TYPE, DIM, ReadMode> texRef;
```

- **TYPE: a texel típusa**
 - INT, FLOAT (ill. 1/2/4 komponensű vektor típusok)
- **DIM: a textúra dimenziója**
 - 1, 2 vagy 3
- **ReadMode: az olvasás során visszaadott adattípus**
 - cudaReadModeElementType: az olvasott adat a texel típusának megfelelő
 - cudaReadModeNormalizedFloat: automatikus konverzió normalizált lebegőpontos értékre → [0.0, 1.0] ill. [-1.0, 1.0] tartományra

Textúra paraméterek

- **Címzési mód**

- Normalizált: a textúra texelei a $[0, 1]$ tartományba eső értékekkel címezhetők
- Nem-normalizált: a texelek a textúra szélességnek és magasságnak megfelelő egész értékekkel címezhetők

- **filterMode**

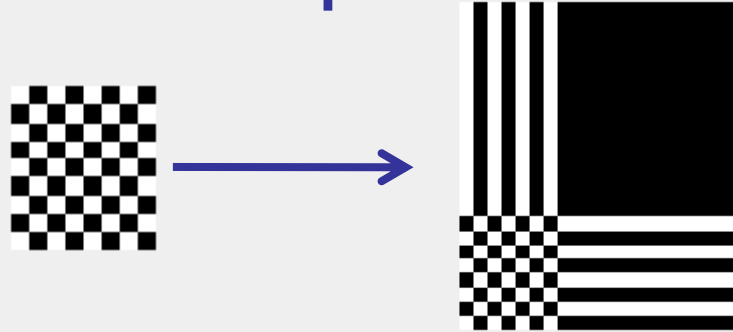
- cudaFilterModePoint: nem texel középpontra eső cím esetén a legközelebbi texelt adja vissza
- cudaFilterModeLinear: lineáris interpoláció a legközelebbi 2/4/8 texel használatával (csak FP ReadMode és norm. cím esetén)

- **channelDesc: textúra adattípus megadás**

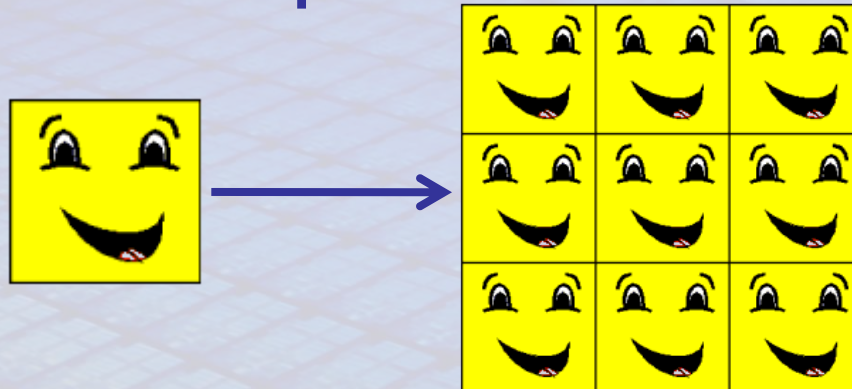
- cudaChannelFormatKindSigned: előjeles egész
- cudaChannelFormatKindUnsigned: pozitív egész
- cudaChannelFormatKindFloat: lebegőpontos

Textúra címzési módok

- `cudaAddressModeClamp`



- `cudaAddressModeWrap`



- `cudaAddressModeMirror`: mint a Wrap, csak tükrözve
- `cudaAddressModeBorder`: kicímzés esetén 0-t ad vissza

Textúra példa – hoszt (1.)

```
texture<float, 2, cudaReadModeElementType> tex0;
```

Referencia

```
.....
```

```
int main(){
```

```
    float *hA, *hB;
```

```
    hA = (float*)(malloc(size)); hB = (float*)(malloc(size));
```

```
    for (int i=0; i<256*256; i++) hA[i] = float(i);
```

```
    cudaChannelFormatDesc channelDesc =
```

```
        cudaCreateChannelDesc(32, 0, 0, 0,  
        cudaChannelFormatKindFloat);
```

**Komponens
leíró**

```
    cudaArray* cuArray;
```

**GPU
memória**

```
    cudaMallocArray(&cuArray, &channelDesc,  
        256, 256);
```

**Textúra memória
foglalás**

Textúra példa – hoszt (2.)

```
.....  
tex0.addressMode[0] = cudaAddressModeWrap;  
tex0.addressMode[1] = cudaAddressModeWrap;  
tex0.filterMode      = cudaFilterModePoint;  
tex0.normalized      = true;
```

**Paraméter
beállítás**

```
cudaBindTextureToArray(tex0, cuArray,  
                        channelDesc);
```

**Referencia –
memória
összerendelés**

```
cudaMemcpyToArray(cuArray, 0, 0, hA,  
                  size, cudaMemcpyHostToDevice);
```

**Textúra
memória írása**

```
kernel <<<thrGrid, thrBlock>>>(gB);
```

**Kernel
hívás**

Textúra példa – kernel

- Textúra olvasás a kernelben
 - tex1D(texRef, float x);
 - tex2D(texRef, float x, float y);
 - tex3D(texRef, float x, float y, float z);
- Kernel a példához:

```
__global__ void kernel(float *B)
{
    int thID = blockDim.x * blockIdx.x + threadIdx.x;
    B[thID] = tex2D(tex0, float(threadIdx.x),
                    float(blockIdx.x)) * 2;
}
```

Surface

- CC $\geq$ 2.0 GPU-k esetén a textúra memória írható is, ehhez surface-t kell deklarálni (NEM koherens!)

- Surface referencia deklaráció (hasonló a textúrához)

```
surface<void, cudaSurfaceType2D> outputSurface;
```

- Hoszt: cuArray foglalás: írható

```
cudaMallocArray(&cuArraySrc, &channelDesc,  
src_x, src_y, cudaArraySurfaceLoadStore);
```

- Hoszt kód: cuArray – surface binding

```
cudaBindSurfaceToArray(outputSurface, cuArraySrc)
```

- Kernel kód: surface írási/olvasási függvények

```
float data;  
surf2Dread(&data, inputSurfObj, x * 4, y);  
surf2Dwrite(data, outputSurfObj, x * 4, y);
```

- Címzés byte-os!!!

Optimalizálás

- **Hoszt – GPU átvitel**

- 4 – 12 Gbyte/s → relatíve lassú
- Nagyobb mennyiségű adat átvitele hatékonyabb
- Nem-lapozható memória gyorsabb
- Kernel hívások közötti átmeneti adatok másolása felesleges
- Memória másolás és végrehajtás átlapolható

- **GPU globális (külső) memória**

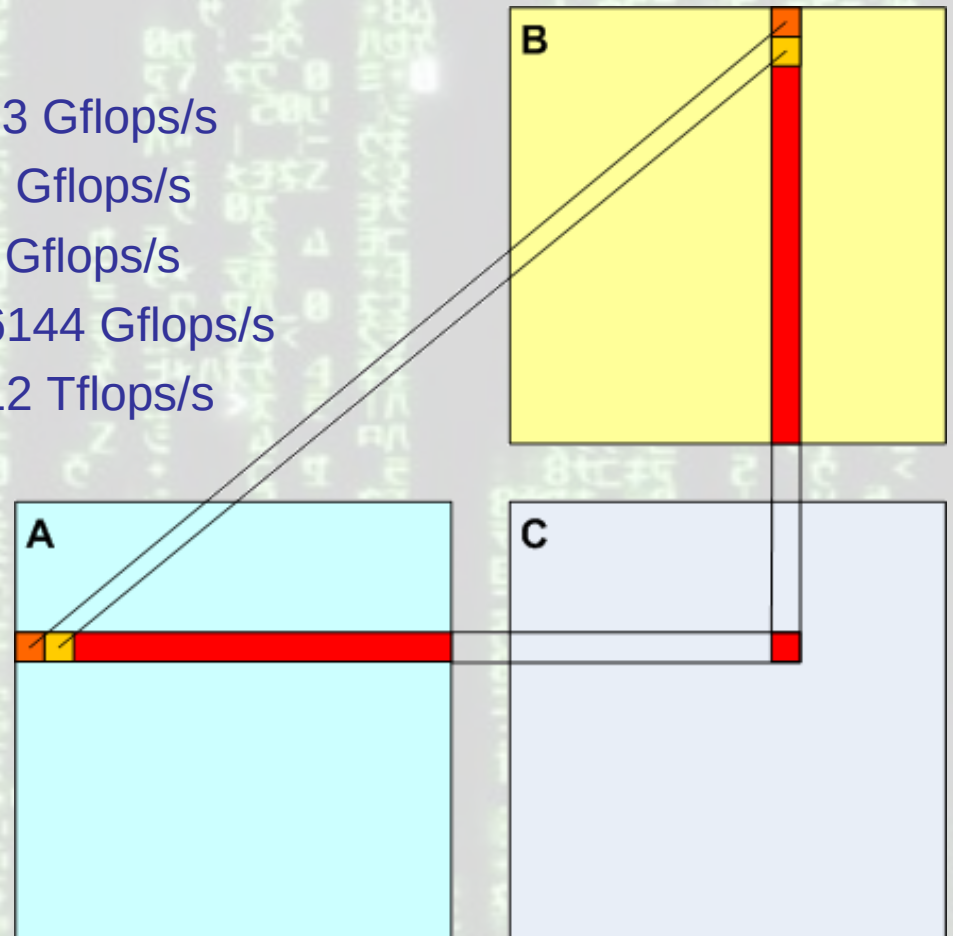
- Viszonylag nagy sávszélesség (>100 Gbyte/s)
- Sokszor ez a limit
- Általában érdemesebb valamit többször kiszámolni, mint eltárolni és visszaolvasni

Kernel operátorok

- **Szinkronizáció Thread Block-on belül**
 - `__syncthreads()`
- **Matematikai függvények**
 - ULP (unit in the last place): ~különbség a lebegőpontos szám és a legközelebbi valós szám között → legpontosabb: 0,5 ULP
 - `+`, `*`: 0,5 ULP
 - FMAD: Nincs csonkolás a szorzás után!
 - `/`, `sin`, `cos`, `tan`, `log`, `exp`: 3 – 4 ULP (sztenderd könyvtár)
 - `__sin`, `__cos`,: pontatlanabb, gyorsabb
- **Ciklus kifejtés: `#pragma unroll`**
 - Paraméter nélkül vagy 1 vagy részleges kifejtés
 - Paraméterrel: függetlenül a tényleges iteráció számtól, a paraméternek megfelelő számú kibontás!!

Mátrix szorzás

- Két, NxN-es mátrix szorzata
 - $N \times N \times N$ MADD művelet
 - $N \times N \times N \times 2$ load, $N \times N$ store
- GPU-k
 - GF100: 1600 MHz, 448 ALU, 1433 Gflops/s
 - GF104: 1350 MHz, 336 ALU, 907 Gflops/s
 - GK107: 1100 MHz, 384 ALU, 845 Gflops/s
 - GP104: ~1600 MHz, 1920 ALU, 6144 Gflops/s
 - GP102: ~1600 MHz, 3840 ALU, 12 Tflops/s



MMUL: C kód (1)

■ Emlékeztető: C / Vektorizált kód

```
for (row=0; row<N; row=row+1) {  
    for (col=0; col<N; col=col+1) {  
        sum = 0;  
        for (i=0; i<N; i++) {  
            sum = sum + *(a_base+N*row+i) * *(b_base+col+i*N);  
        }  
        *(res_base + row*N + col) = sum;  
    }  
}
```

```
.....  
b_br      = vld1q_dup_f32(b_addr);  
m_res[0] = vmulq_f32(a_sse[0], b_br);  
m_res[1] = vmulq_f32(a_sse[1], b_br);  
m_res[2] = vmulq_f32(a_sse[2], b_br);  
m_res[3] = vmulq_f32(a_sse[3], b_br);  
r_sse[0] = vaddq_f32(r_sse[0], m_res[0]);  
r_sse[1] = vaddq_f32(r_sse[1], m_res[1]);  
r_sse[2] = vaddq_f32(r_sse[2], m_res[2]);  
r_sse[3] = vaddq_f32(r_sse[3], m_res[3]);  
a_addr = a_addr + N;  
.....
```

MMUL: C kód (2)

■ CPU implementáció eredmények: Gflops/s

	C kód	C kód + OpenMP	Vektorizált (SSE/NEON)	Vektorizált + OpenMP
Intel Core i5 9300H (4 core, 8 thr; 2,4...4 GHz)	1,4	3,4	10,8	45
AMD Ryzen 5 3600 (6 core, 12 thr; 3,6...4,2 GHz)	0,35	3,9	17,2	105
NVIDIA Tegra X1 (4x ARM A53; 1,73 GHz)	0,17	0,55	0,6	1,71
Qualcomm Snapdragon 820 (4x Kryo; 2,2 GHz)	0,21	0,71	0,63	2,11
NVIDIA AGX Xavier (8x Carmel; 2,3 GHz)	0,56	3,9	2,1	12,6

- Megj.: az ARM Cortex A78 magok egy szálás teljesítménye kb. a Carmel duplája. Az Apple M1 magok 3-3,5x sebességre képesek.

MMUL: GPU (1)

- Minden szál egy kimenetet számít
- 16*16 Thread Block, N/16*N/16 Grid

```
__global__ void kernel_g1(float *A, float *B, float *C)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;

    #pragma unroll 32
    for (int i = 0; i < N; i++)
    {
        sum += A[row*N+i] * B[i*N+col];
    }
    C[row*N+col] = float(sum);
}
```

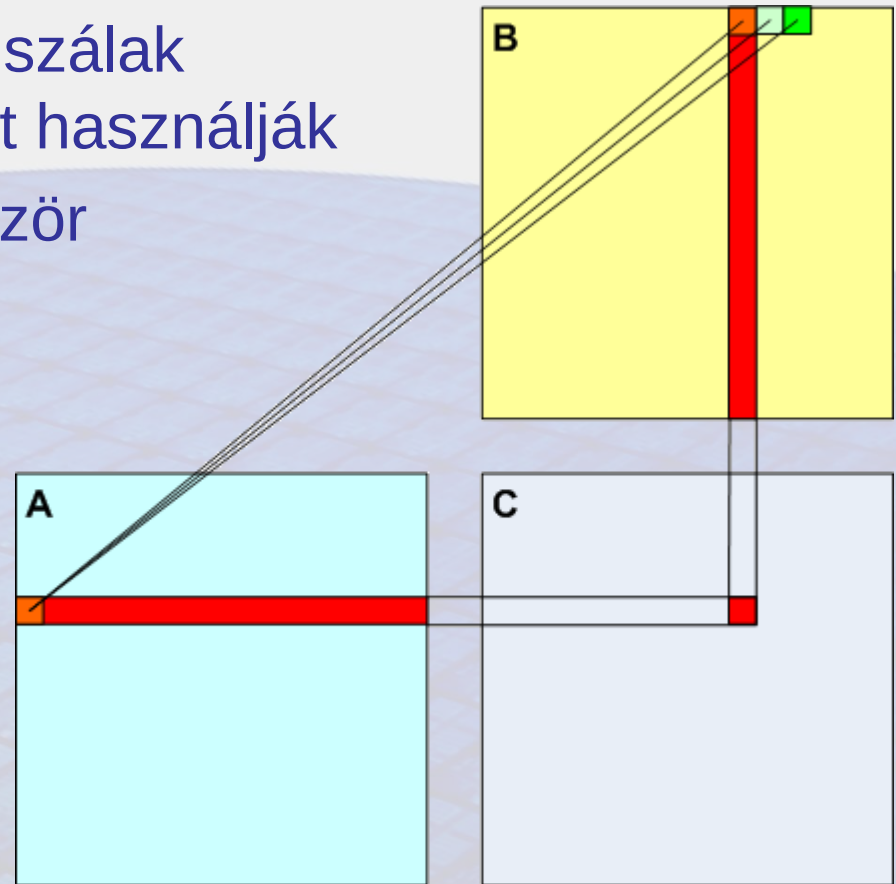
GF104: 75 GF/s, 8%
GF100: 120 GF/s, 8%
GP104: 330 GF/s, 5,4%
GP102: 330 GF/s, 4,3%
X1: 21 GF/s, 4%
AGX: 200 Gf/s, 12%

MMUL: GPU (2)

- **Az előző megoldás sávszélesség limitált**
 - Egy szál a B mátrix egy-egy oszlopán lépked végig → nem burst-ös a memória hozzáférés (cache segít)
 - Egy adott C soron dolgozó szálak ugyanazokat az A elemeket használják
 - Mégis beolvassuk többször



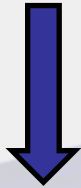
- **Shared Memory**



MMUL: GPU (3)

- **16x16 Thread Block C 16x16 részével foglalkozik**

- B-ből 16 széles részt kell beolvasni
- A-ból a memória burst méretnek megfelelőt $\rightarrow 16$

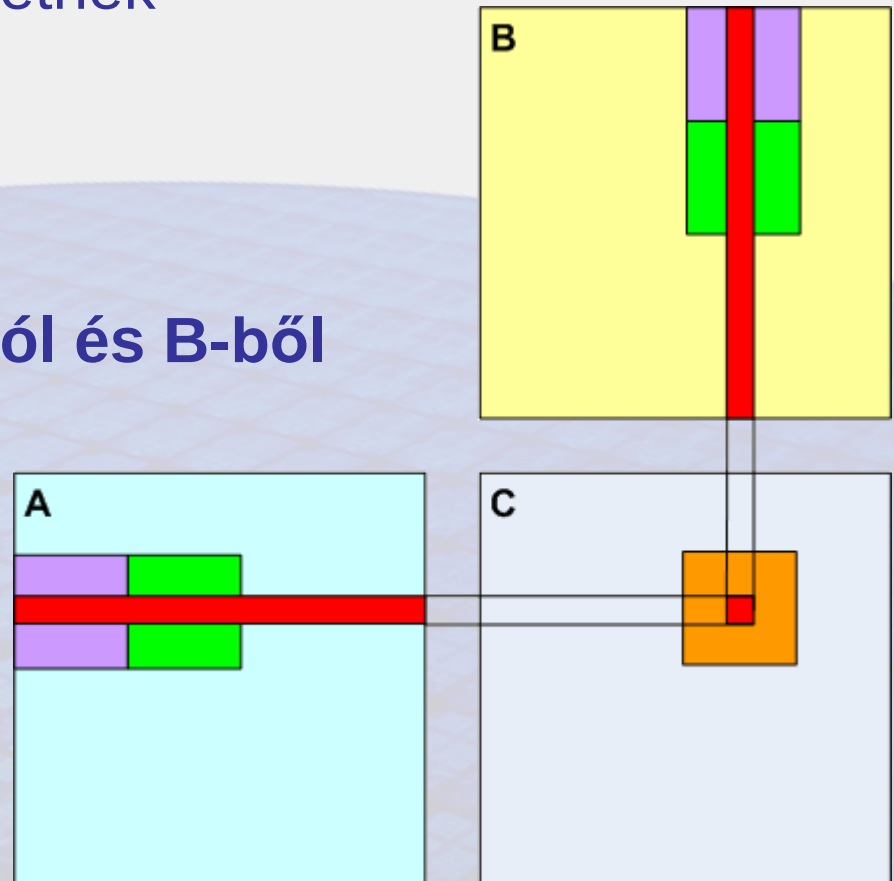


- **16x16 blokkot olvasunk A-ból és B-ből**

- $16 \times 16 \times 4 \times 2 = 2$ kbyte SMEM



- Minden szál egy A és egy B elemet tölt be blokkonként



MMUL: GPU SMEM (1)

```
for (int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep)
{

    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    As[ty][tx] = A[a + N * (ty) + tx];
    Bs[ty][tx] = B[b + N * (ty) + tx];

    tx, ty: threadIdx.x, y
    bx, by: blockIdx.x, y

    __syncthreads();
}
```

**16x16 blokkonként
lépkedünk végig**

**Shared Memory
deklaráció**

**Minden szál A és B
egy-egy elemét tölti
be**

**A szálak bevárják
egymást → a két
memória minden
eleme be van töltve**

MMUL: GPU SMEM (2)

```
.....  
#pragma unroll  
for (int k = 0; k < 16; k=k+1)  
{  
    float b_reg = Bs[k][tx];  
    Csub += As[ty][k] * b_reg;  
}
```

**16x16-os blokkon belül
minden szál 16 MAC műveletet
végez**

```
__syncthreads();  
}
```

**Szálak bevárják egymást az
újabb SMEM töltés előtt**

```
int c = N * 16 * by + 16 * bx;  
C[c + N*(ty) + tx] = csum;
```

GF104: 125 Gf/s, 14%
GF100: 250 Gf/s, 17%
GK107: 71 Gf/s, 8%
GP104: 980 Gf/s, 16%
GP102: 1887 Gf/s, 15%
X1: 61 Gf/s, 12%
AGX: 383 Gf/s, 27%

MMUL: GPU SMEM (3)

- **GeForce GTX 460 (900 Gflops/s)**
 - Külső memóriás verzió: 75 Gflops/s → 8%
 - Shared Memory: 125 Gflops/s → 14%
 - **Limit:**
 - Shared Memory sávszélesség
 - 32 bank, 1 port/bank  **16 olvasás / ALU clk**
 - ALU órajel felével jár (Fermi)
 - És csak 32 ALU-t használunk a 48-ból
- 
- **Egy szál számoljon több kimenetet → blokkméret: 32*32**
 - Közös értéket elegendő egyszer kiolvasni SMEM-ből
 - Egymástól függetlenül számíthatók → ILP

MMUL: 2 kimenet / szál

- Minden szál 2-2 elemet tölt SMEM-be

```
As[ty+0][tx] = A[a + N * (ty+0) + tx];  
Bs[ty+0][tx] = B[b + N * (ty+0) + tx];  
As[ty+16][tx] = A[a + N * (ty+16) + tx];  
Bs[ty+16][tx] = B[b + N * (ty+16) + tx];
```

- Két akkumulátor regiszter

```
float b_reg = Bs[k][tx];  
csum[0] += As[ty+0][k] * b_reg;  
csum[1] += As[ty+16][k] * b_reg;
```

- 32*16 szál / Thread Block
 - 25 regiszter / szál → Occ.: 67%

GF104: 180 Gf/s, 20%
GF100: 392 Gf/s, 24%
GK107: 102 Gf/s, 12%
GP104: 1560 Gf/s, 25%
GP102: 2917 Gf/s, 24%
X1: 108 Gf/s, 22%
AGX: 572 Gf/s

MMUL: 4-8 kimenet / szál

- **Továbbra is 32x32 kimeneti blokk**
 - 4 kimenet: 32x8 Thread Block
 - 8 kimenet: 32x4 Thread Block
- **Kihasználtság (occupancy)**
 - 32 regiszter/szál → 67% occup.
 - 47 regiszter/szál → 42% occup.
- **További kimenet/szál növelés ezen a módon nem jó**
 - Thread Block mérete nagyon lecsökken
 - 32x2 → 64 thread
 - Vagy kimeneti blokkméret növelés: 64x64
 - (64x64x4)x2 → 32768 byte → 1 Thread Block/SM

GF104: 225 Gf/s, 25%
GF100: 492 Gf/s, 34%
GK107: 132 Gf/s, 16%
GP104: 1960 Gf/s, 32%
GP102: 3594 Gf/s, 29%
X1: 144 Gf/s, 29%
AGX: 740 Gf/s, 52%

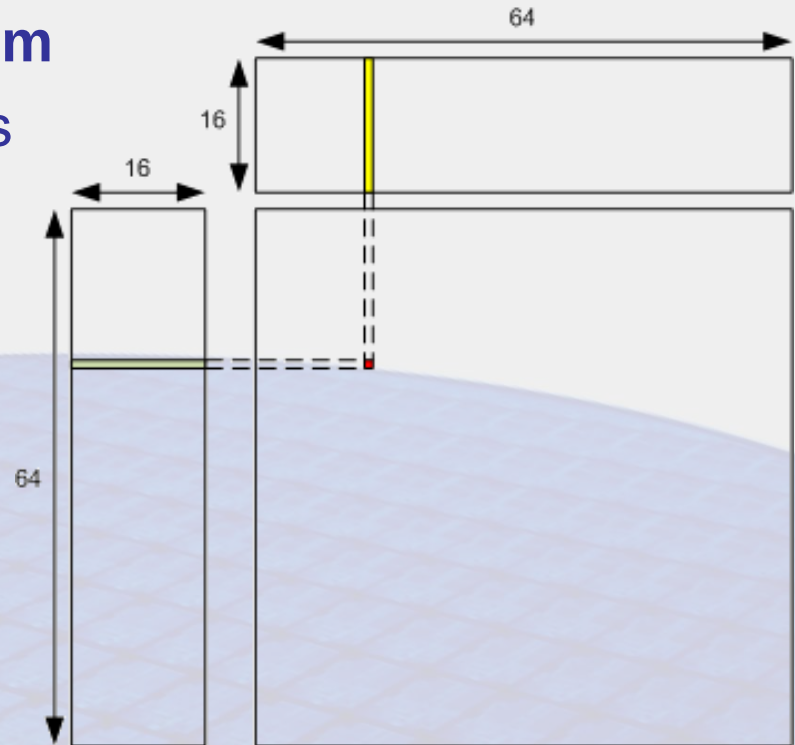
GF104: 260 Gf/s, 29%
GF100: 553 Gf/s, 39%
GK107: 154 Gf/s, 18%
GP104: 2250 Gf/s, 37%
GP102: 4297 Gf/s, 35%
X1: 165 Gf/s, 33%
AGX: 820 Gf/s, 57%

MMUL 3. (1)

- Az eddigi megoldások sávszélesség limitáltak
- Bár növeltük az egy szál által végzet számítások mennyiségét, drasztikusan nem változott a sávszélesség igény – MAC műveletek aránya
- Jó megközelítés: töltsünk be regiszterbe néhány értéket, majd végezzünk el annyi MAC műveletet, amennyit ezekkel lehet
 - Ehhez is szálanként több akkora van szükség

MMUL 3. (2)

- **Legyen egy Thread Block-hoz tartozó kimeneti (C) blokkméret 64x64 elem**
 - B-ből minden ciklusban 64 széles tartományt töltünk be
 - A-ból minimum 16 széleset (64 byte burst), és B miatt 64 mélyet
 - A SMEM: $16 \times 64 = 4$ kbyte
 - B SMEM: $64 \times 16 = 4$ kbyte
- **Thread block: $16 \times 16 \rightarrow$ minden szál 4×4 kimenetet számít**
 - Mind x, mind y irányban 16-os lépésközzel $\rightarrow$ azaz az egymás melletti 16 érték egymást követő 16 szálhoz tartozik (ld. következő ábra)

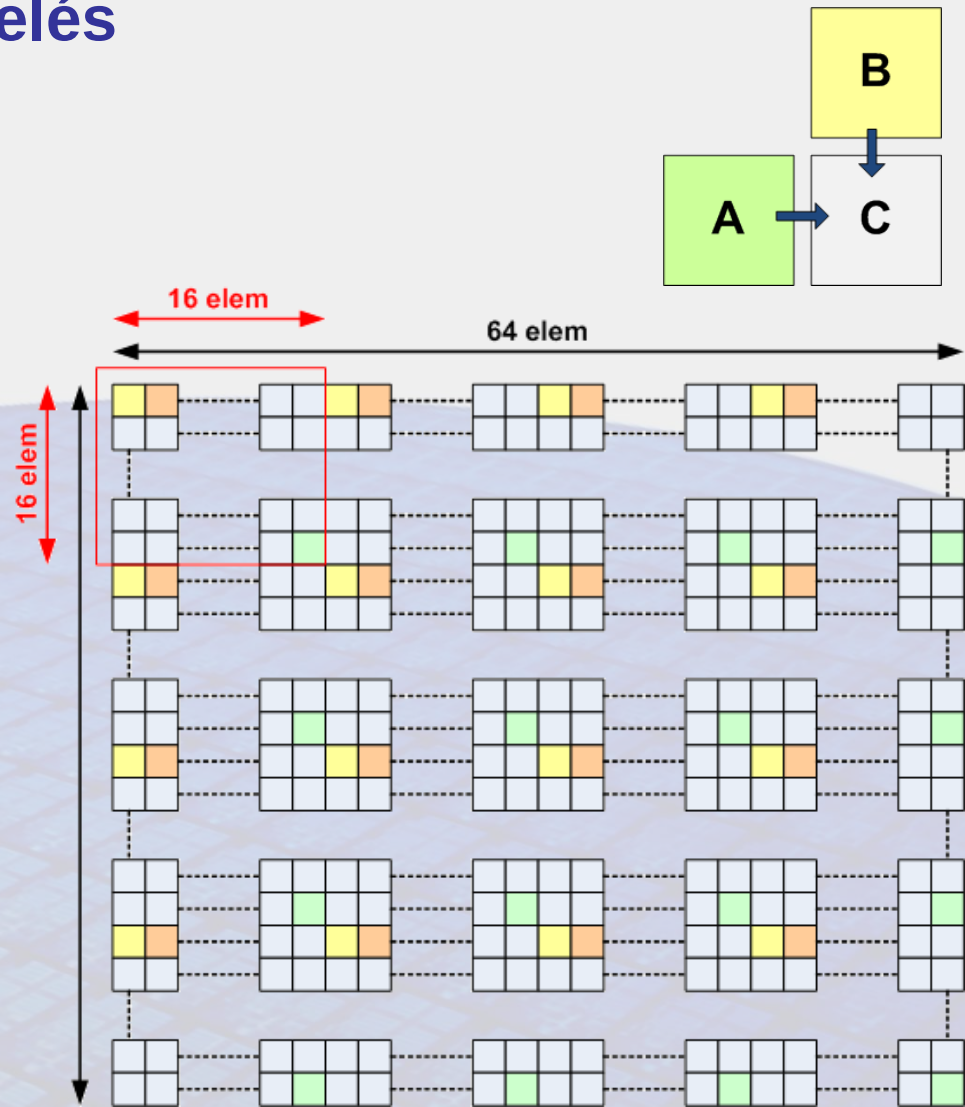


MMUL 3. (3)

- **Kimenet – szál összerendelés**

- Az azonos színezett elemeket azonos szál számítja

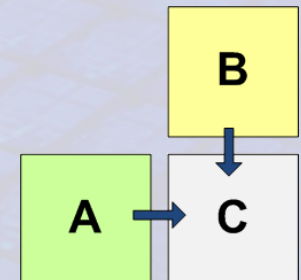
- A mtx 1x64 rész-oszlopa és B mtx 64x1 rész-sora alapján minden elem 1-1 rész-szorzata számítható
- SMEM 16 oszlopot (A) és 16 sort (B) tartalmaz



MMUL 3. (4)

- **A teljes mátrixra**
 - A 16x64 rész-mátrixának betöltése
 - 2 sor / warp → nincs bank konfliktus
 - B 64x16 rész-mátrixának betöltése
 - Warp 32 szálának egy soron belül 32 egymást követő elemet kell tölteni
- **A betöltött adatokra (16x)**
 - A egy oszlopából 4 érték regiszterbe töltése
B egy sorából 4 érték regiszterbe töltése
 - A 4-4 érték alapján 16 rész-szorzat számítása
- **Kihasználtság: 33%**
 - 52 regiszter/thread
 - 8kB SHMEM

GF104: 400 Gf/s, 40%
GF100: 765 Gf/s, 53%
GK107: 230 Gf/s, 27%
GP104: 3350 Gf/s, 54%
GP102: 5750 Gf/s, 47%
TU106: 2720 Gf/s, 50%
X1: 250 Gf/s, 50%
AGX: 1100 Gf/s, 75%



MMUL 3. (5)

- **A MAC és LOAD műveletek aránya tovább javítható**
 - Még több elemet töltünk A-ból és B-ből
 - Képezzük a betöltött elemek szorzatát, akkumulálunk
- **Növeli a regiszter igényt**
 - Több bemeneti regiszter
 - Több akkumulátor regiszter
- **Pl.**
 - 16x8 ThreadBlock
 - Egy ThreadBlock $(16 \times 8) \times (8 \times 8)$ elemet számol $\rightarrow$ 64 kimenet/szál
 - 128×16 (A) és 16×64 (B) Shared Memory
 - Egy szál egy iterációban
 - $8 + 8$ elemet tölt be A-ból, B-ből
 - $8 \times 8 = 64$ részsorzat, 64 akkumulátor
 - Kb 96-128 regiszter/szál

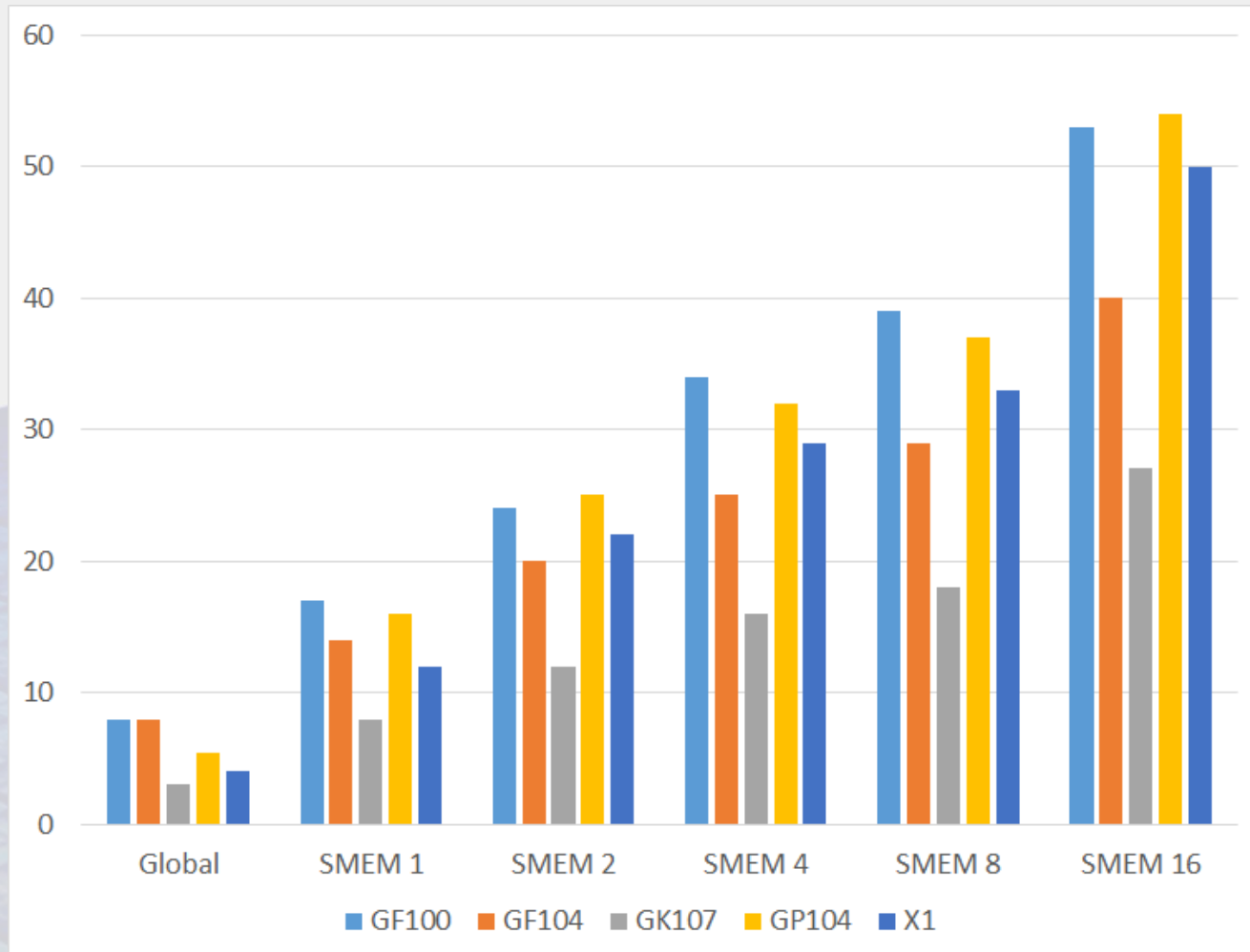
GP102: 7900 Gf/s, 65%

TU106: 3600 Gf/s, 67%

AGX: 1244 Gf/s, 87%

Mmul teljesítmény (%)

- Elvi számítási teljesítmény %-ában

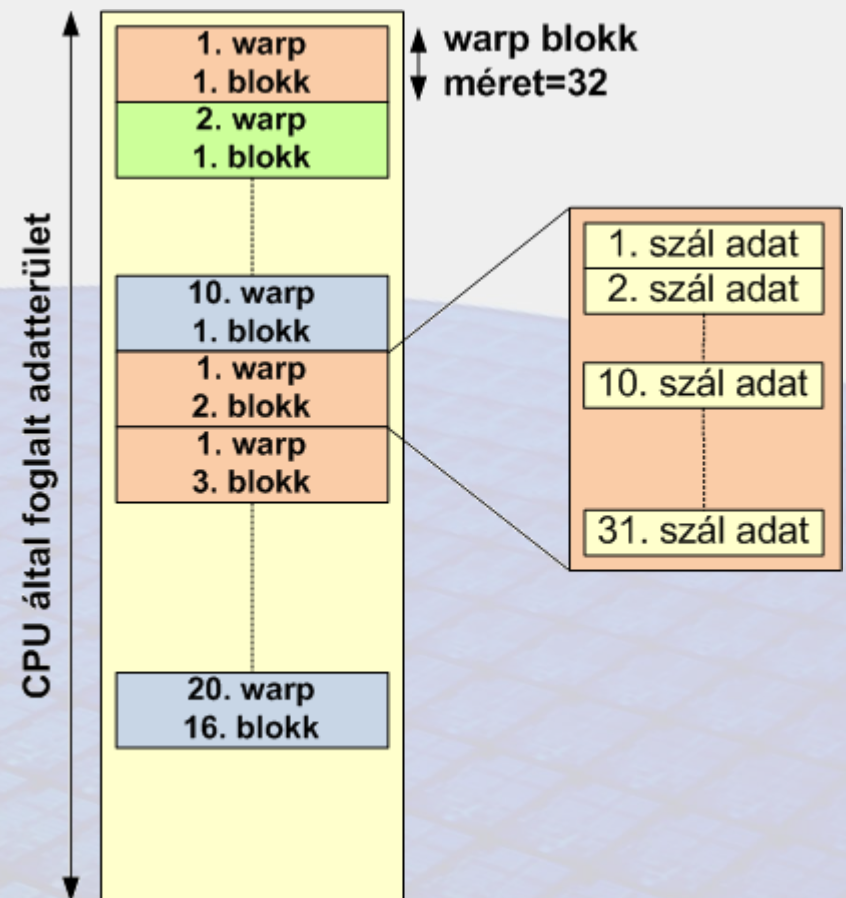


Ismeretlen kimeneti adatméret

- A feladatok jelentős részénél a bemeneti adatmennyiség egyértelműen meghatározza a kimeneti adatmennyiséget (pl. $mtx \cdot mul$, képfeldolgozás, FFT, ...) → OK
- Egyes esetekben a szálak által generált kimeneti mennyiség nem determinisztikus
 - CPU egyetlen memória területet foglal (remélt worst-case méret)
 - Warp-ok allokálnak egy-egy új blokkot
 - Coalesced write
 - Szálak ezen belül dinamikusan allokálnak egy-egy új blokkot

Dinamikus blokk v1

- Ha szükséges, a warp új blokkot foglal
- A warp minden szála ír
 - Érvényes adatot vagy
 - „Nem érvényes” szót
 - Coalesced hozzáférés
- **Blokk méret = warp méret**
 - Vagy többszöröse
- **Rossz lehet a memória kihasználtság**
 - Worst case: 1/32

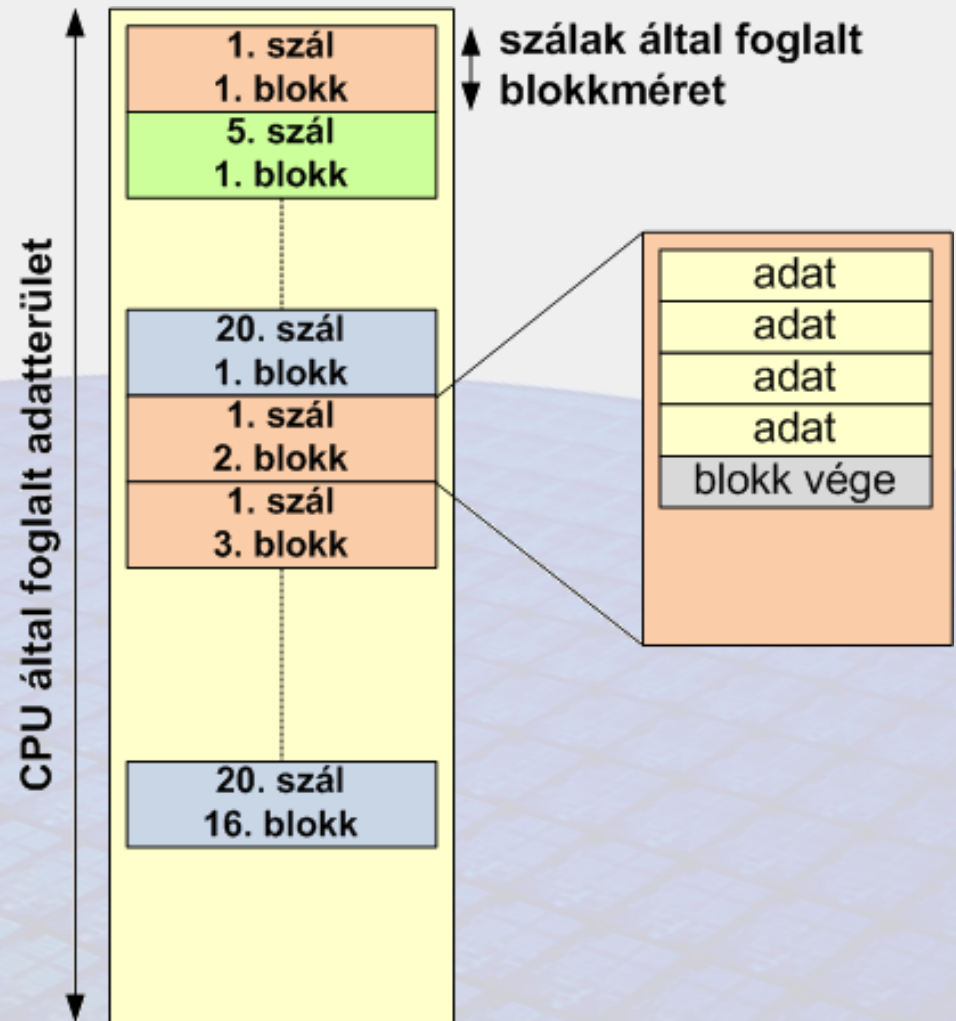


Dinamikus blokk v1

- **Global Memory-ban**
 - A következő szabad memória blokk címét tartalmazó szó
- **SMEM-ben a thread block minden warp-jára**
 - 1 kimenet érvényes regiszter (out_valid)
 - Globális memória blokk cím (gl_addr)
- **Egy warp minden szála**
 - out_valid-t növeli egyel ha van érvényes adata
 - AtomicAdd() SMEM-ben
 - Ha out_valid != 0
 - A warp 0. szála növeli a szabad memória blokk címet
 - AtomicAdd() Global Memory-ban
 - A szálak kiírják adatukat
 - Cím: Memória blokk kezdőcím (gl_addr) + (threadIdx & 0x1F)

Dinamikus blokk v2

- A blokkok a szálakhoz tartoznak
- Teljes memóriaterület
 - A lehető legnagyobb
- Blokkméret
 - 1: gyakori foglалás
 - Túl nagy: pazarló
 - ~átlagos kimeneti szám
- GPU → PC átvitel
 - Csak a használt méret

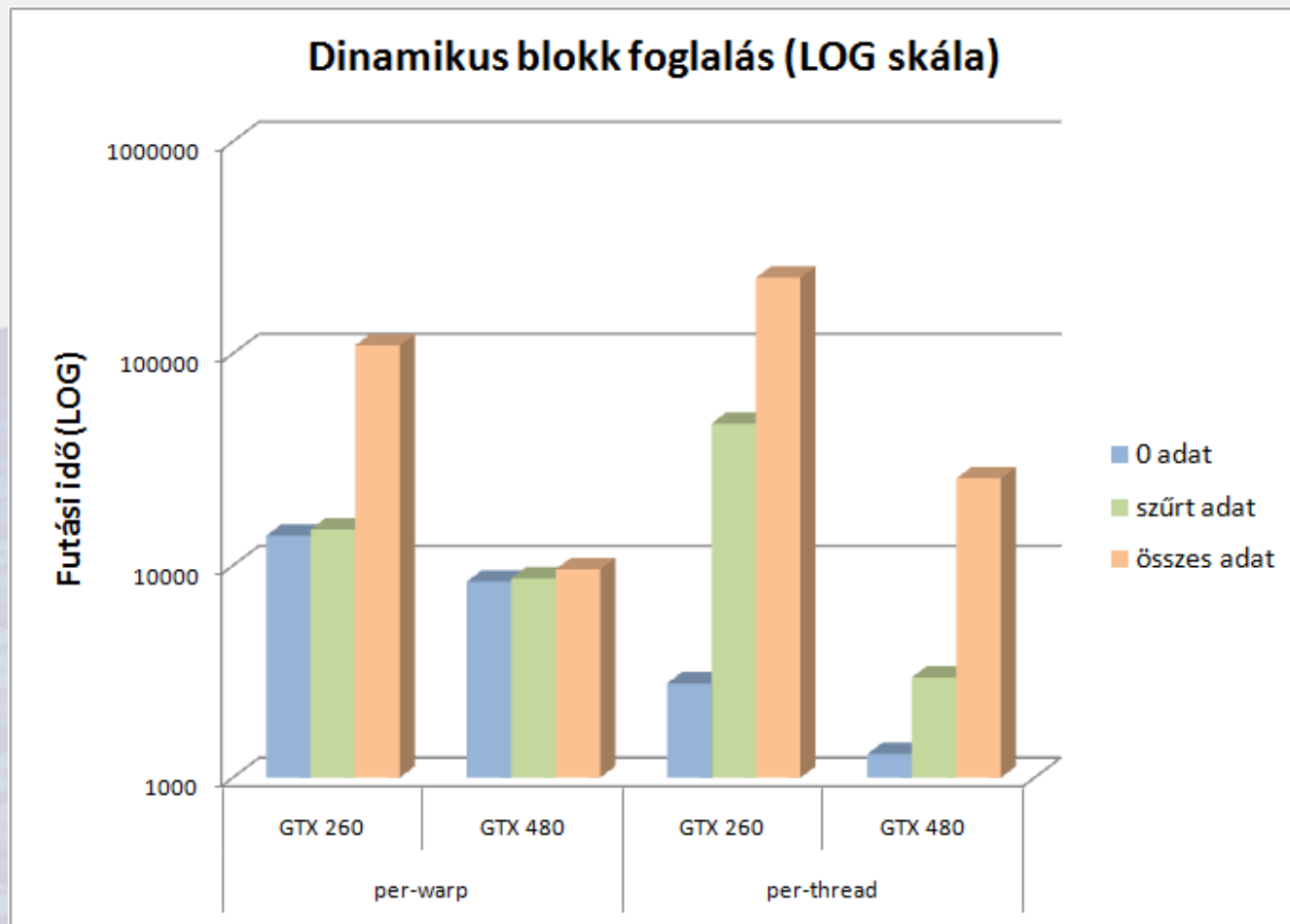


Blokk foglалás – global atomic

- A globális memória tartalmaz egy szót, mely az első szabad blokk kezdőcímét tartalmazza
 - Inicializálás: 0
- Ha egy szálnak új blokkra van szüksége, ez lesz annak a címe
 - A szabad blokk címet pedig növeli a blokkmérettel
 - Ehhez atomi utasításra van szükség → AtomicAdd()
 - Globális memória atomi művelet lassú (Kepler óta annyira azért nem)
 - Túl kicsi blokkméret esetén túl gyakori
- Hátrány: blokkon belül az egymást követő címeket ugyanaz a szál írja → nem coalesced
 - Cache itt rengeteget segít

Sebesség

- GTX 260: CC1.3 (nem cache-elt)
- GTX 480: CC2.0 (cache-elt)



Stream

- **Stream: parancsok sorozata**
 - Alapértelmezett stream: 0
- **Konkurrens hoszt – GPU végrehajtás**
 - CudaMemcpyAsync();

```
CudaMemcpyAsync ( ... ) ;  
kernel <<< ... >>> ( ... ) ;  
HostFunction ( ... )
```

- **Egyes GPU-kkal lehetséges a memória műveletek és a kernel futtatás párhuzamosítása**
 - Nem 0. stream
 - Nem lapozható memória
 - deviceOverlap státusz

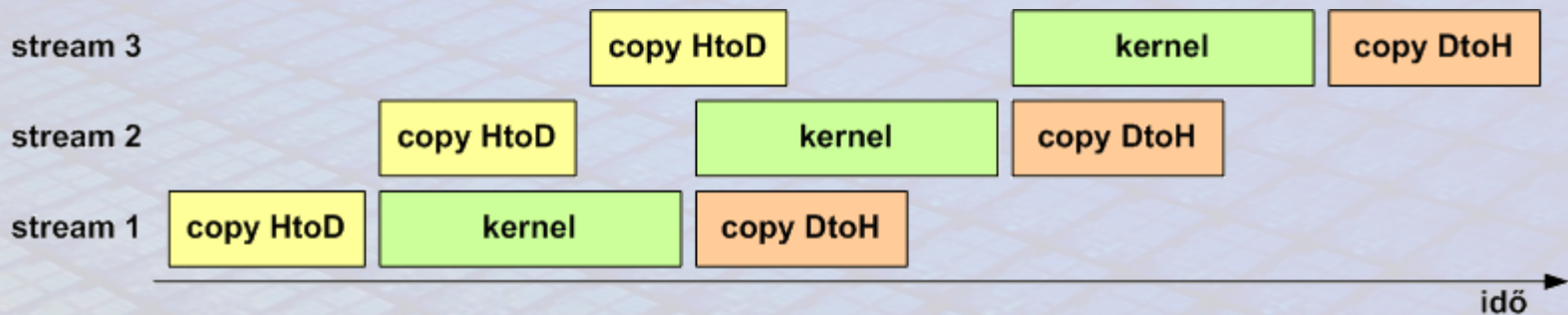
Stream – adatmozgatás

- Tipikus feldolgozás: másolás → kernel → másolás



- Stream: másolás és kernel futtatás időben átlapolható

- Nem minden GPU támogatja
- Fermi alapú Tesla: 2 DMA egység a két irányú adatmozgatáshoz



Stream – konkurrens GPU

- Egy Stream-en belül a végrehajtás szekvenciális
- A Stream-ek egymással párhuzamosan hajtódnak végre
- Tipikus stream:

```
CudaMemcpyAsync ( ... ) ;  
kernel <<< ... >>> ( ... ) ;  
cudaMemcpyAsync ( ... ) ;
```
- `cudaThreadSynchronize()`: minden stream végrehajtása befejeződött
- Stream generálás: `cudaStreamCreate( )`
- Stream felszabadítás: `cudaStreamDestroy( )`

Stream példa

```
cudaStream_t stream[2];  
for (int i = 0; i < 2; ++i)  
    cudaStreamCreate(&stream[i]);  
  
for (int i = 0; i < 2; ++i)  
    cudaMemcpyAsync(inGPU+i*N, inCPU+i*N, N,  
                    cudaMemcpyHostToDevice, stream[i]);  
  
for (int i = 0; i < 2; ++i)  
    kernel<<<100, 512, 0, stream[i]>>>(outGPU+i*N,  
                                         inGPU+i*N, N);  
  
for (int i = 0; i < 2; ++i)  
    cudaMemcpyAsync(outCPU+i*N, outGPU+i*N, N,  
                    cudaMemcpyDeviceToHost, stream[i]);  
cudaThreadSynchronize();
```

N: memória méret
inGPU, outGPU: GPU
memória
inCPU, outCPU: hoszt
memória

hoszt → GPU
átvitel 2 stream-
ben

kernel futtatás

GPU → hoszt
átvitel 2 stream-
ben

Stream – várható nyereség

- **1 Stream**

- Másolás:

- Host to Device: $H2D = 16 \text{ ms}$

- Device to Host: $D2H = 16 \text{ ms}$

- Kernel futási idő: $K = 32 \text{ ms}$

- Teljes futási idő: $H2D + K + D2H = 16 + 32 + 16 = 64 \text{ ms}$

- **4 stream → stream-enként $H2D/4$, $D2H/4$ és $K/4$**

- Futási idő: $H2D/4 + (K/4)*4 + D2H/4 = 4 + 32 + 4 = 40 \text{ ms}$

PTX assembly

- „Virtuális utasításkészlet”
- A C/C++ kódok PTX kóddá fordulnak, majd a PTX kód fordul gépi tényleges gépi kóddá
- Van lehetőség PTX assembly betétek elhelyezésére a C kódban

```
asm(  
  „assembly kód” :  
  „constraint”(kimenet) :  
  „constraint”(bemenet)  
  ) ;
```

- Nem igazi assembly, regiszter allokáció és némi optimalizáció is követi

PTX példa

- **Pl. összeadás:**

```
asm(  
    „add.s32 %0, %1, %2;” :  
    „=r”(q) :  
    „r”(x), „r”(y)  
    );
```

Assembly utasítás

Kimenet: regiszter

Bemenet: 2 regiszter

- **Regiszter deklaráció ASM kódban:**

```
asm(  
    „.reg .s32 t0;\n\t”  
    „add.s32 t0, %1, %2;\n\t”  
    „add.s32 %0, t0, %3;” :  
    „=r”(q) :  
    „r”(x), „r”(y), „r”(z)  
    );
```

Ha nem írunk kernelt...

- **cuBLAS: CUDA Basic Linear Algebra Subroutine**
 - Vektor és matrix műveletek
- **NVBLAS:** GPU gyorsított BLAS
- **nvJPEG:** JPEG dekóder
- **cuFFT:** 1D/2D/3D; valós és complex; 16/32/64 bit float
- **nvGRAPH:** lineáris algebra alapú gráf manipuláció
- **cuRAND:** véletlen szám generálás
- **cuSPARSE:** ritka mátrixokon végzett műveletek
- **cuSOLVER:** matrix faktorizáció, least-square, eigen, stb.
- **NPP: NVIDIA Performance Primitives**
 - Jelfeldolgozás, képfeldolgozás, transzformációk, morfológiai operátorok, statisztikai operátorok
- **Thrust: ~Standard Template Library**
 - Rendezés, redukció, particionálás, stb.
- **cuDNN: NVIDIA CUDA Deep Neural Network library**
 - Pl. Caffe, Caffe2, Torch, TensorFlow GPU gyorsítása

Heterogén számítási rendszerek

OpenCL

Szántó Péter

BME MIT, FPGA Laboratórium

OpenCL

- Az alapvető felépítése nagyon hasonló a CUDA-hoz, elsősorban GPU-khoz készült
- Jelenleg számos eszközre van OpenCL driver (GPU-k, FPGA-k, CPU-k, DSP-k)
- x86 vonalon
 - Intel: CPU és integrált GPU
 - AMD: CPU és GPU
 - NVIDIA: GPU

CUDA → OpenCL (1)

- Szál hierarchia, memória elnevezések

CUDA	OpenCL
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

CUDA → OpenCL (2)

- Kernel kulcsszavak

CUDA	OpenCL
<code>__global__</code>	<code>__kernel</code>
<code>__device__</code>	---
<code>__constant__</code>	<code>__constant</code>
<code>__device__</code>	<code>__global</code>
<code>__shared__</code>	<code>__local</code>

CUDA → OpenCL (3)

▪ Szál azonosítók

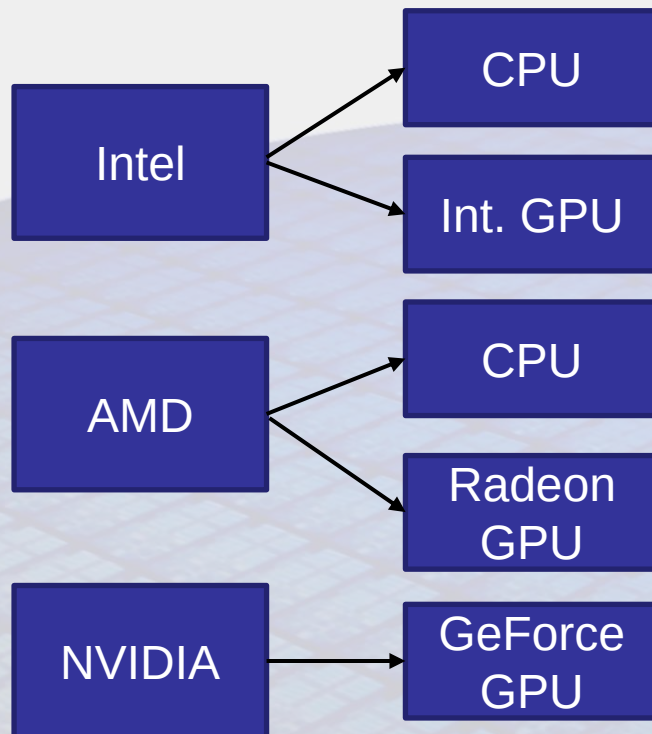
CUDA	OpenCL
threadIdx.x	get_local_id(0)
blockIdx.x	get_group_id(0)
blockDim.x	get_local_size(0);

▪ Szinkronizáció

CUDA	OpenCL
__syncthreads()	barrier()
__threadfence()	N/A
__threadfence_block()	mem_fence(CLK_GLOBAL_MEM_FENCE) mem_fence(CLK_LOCAL_MEM_FENCE)
N/A	read_mem_fence()
N/A	write_mem_fence()

OpenCL platform

- Az egyes eszközök drivere egy ICD (installable client driver)
- Egy ICD → egy platform (lehet több)
- Egy ICD-n belül több eszköz → device
- Pl. Intel CPU-s gépben AMD és NVIDIA GPU:



OpenCL platform & device (1.)

- Platformok és eszközök száma:

```
ret = clGetPlatformIDs(0, NULL, &ret_num_platforms);  
cl_platform_id *platforms;  
platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id)*  
    ret_num_platforms);  
ret = clGetPlatformIDs(ret_num_platforms, platforms,  
    &ret_num_platforms);  
  
int num_devices_all = 0;  
for (int platform_id = 0; platform_id < ret_num_platforms;  
    platform_id++)  
{  
    ret = clGetDeviceIDs(platforms[platform_id],  
        CL_DEVICE_TYPE_ALL, 0, NULL, &ret_num_devices);  
    num_devices_all = num_devices_all + ret_num_devices;  
}
```

OpenCL platform & device (2)

- Platformok és eszközök száma:

```
cl_device_id *devices;
int device_offset = 0;
devices = (cl_device_id*)malloc(sizeof(cl_device_id)*
    num_devices_all);
for (int platform_id = 0; platform_id < ret_num_platforms;
    platform_id++)
{
    ret = clGetDeviceIDs(platforms[platform_id],
        CL_DEVICE_TYPE_ALL, 0, NULL, &ret_num_devices);
    ret = clGetDeviceIDs(platforms[platform_id],
        CL_DEVICE_TYPE_ALL, ret_num_devices,
        &devices[device_offset], &ret_num_devices);
    device_offset = device_offset + ret_num_devices;
}
```

OpenCL context, kernel

- Minden, az OpenCL eszköznek kiadott parancs egy, az eszközhöz létrehozott „context” parancs listájában

```
context = clCreateContext(NULL, 1, &device_id, NULL, NULL,
    &ret);
command_queue = clCreateCommandQueue(context, device_id,
    CL_QUEUE_PROFILING_ENABLE, &ret);
```

- Kernel: (tipikusan szöveg file-ból olvasott) string

- Fordítás: JIT

```
program = clCreateProgramWithSource(context, 1, (const char
    **)&source_str, (const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
    NULL);
kernel = clCreateKernel(program, KERNEL_FUNCTION, &ret);
```

OpenCL kernel paraméterek

- **Kernel deklaráció:**

```
__kernel void kernel_conv_4p(  
    __global unsigned char* gInput,  
    __global unsigned char* gOutput,  
    __constant float *filter_coeffs,  
    int imgWidthF)
```

- **Kernel paraméterek megadása egyesével**

```
ret = clSetKernelArg(kernel, 0, sizeof(device_imgSrc), (void  
    *)&device_imgSrc);  
ret = clSetKernelArg(kernel, 1, sizeof(device_imgDst), (void  
    *)&device_imgDst);  
ret = clSetKernelArg(kernel, 2, sizeof(device_coeffs), (void  
    *)&device_coeffs);  
ret = clSetKernelArg(kernel, 3, sizeof(int), &imgWidthF);
```

OpenCL kernel paraméterek

▪ Memória foglalás:

```
cl_mem device_imgSrc, device_imgDst, device_coeffs;  
device_imgSrc = clCreateBuffer(context, CL_MEM_READ_ONLY,  
    size, NULL, &ret);  
device_imgDst = clCreateBuffer(context, CL_MEM_WRITE_ONLY,  
    size, NULL, &ret);  
device_coeffs = clCreateBuffer(context, CL_MEM_READ_ONLY,  
    5*5*sizeof(float), NULL, &ret);
```

▪ Memória másolás, parancsok végrehajtása

```
ret = clEnqueueWriteBuffer(command_queue, device_imgSrc,  
    CL_TRUE, 0, size, imgSrc, 0, NULL, NULL);  
ret = clEnqueueWriteBuffer(command_queue, device_coeffs,  
    CL_TRUE, 0, 25 * sizeof(float), filter_laplace_f, 0, NULL,  
    NULL);  
  
clFinish(command_queue);
```

OpenCL kernel hívás

- Indított szálak: workgroup és teljes (!!)

```
size_t local_size[] = { 16, 16 };  
size_t global_size[] = { imgWidth, imgHeight};
```

- Kernel futtatás

```
ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,  
    global_size, local_size, 0, NULL, &event);
```

Mért csúcsteljesítmény

	Elméleti telj. (Gflops/s)	char Gops/s	int Gops/s	float Gflops/s
Intel Core i7 960 Intel OpenCL	~105	24	52	78
Intel Core i7 960 AMD OpenCL	~105	16	18	19
Snapdragon 820 NEON + OpenMP	~60	95	27	48
Tegra X1 NEON + OpenMP	~54	106	26	42
Qualcomm Adreno 530	~450	120	63 (int24: 180)	240
NVIDIA Tegra TX1 GPU	~500	231	164	466
NVIDIA GP104 CUDA	~7000	2466	2376	6600
NVIDIA GP104 OpenCL	~7000	2655	2273	5547
AMD Radeon 7750	~820	134	162	796
Intel HD Graphics 4000	~300	120	63	240

OpenCL/CUDA futási idők

	Elméleti telj. (GF/s)	Mátrix szorzás (csak kernel idő!)			
Qualcomm Snapdragon 820	~60	2,66 Gflops/s OpenMP + NEON			
NVIDIA Cortex A53	~54	2,55 Gflops/s OpenMP + NEON			
Intel Core i7 960	~105	16,5 Gf/s OpenMP + SSE	32 Gf/s OpenMP+ SSE blokkos	41 Gf/s Intel OpenCL	3,42 Gf/s AMD OpenCL
Qualcomm Adreno 530	~450	43,7 Gflops/s OpenCL			
NVIDIA Tegra TX1 GPU	~500	223 Gflops/s CUDA			
NVIDIA GP104	~7000	3350 Gflops/s CUDA		3044 Gflops/s OPENCL	
AMD Radeon 7750M	~820	426 Gflops/s OpenCL			
Intel HD Graphics 4000	~300	76 Gflops/s OpenCL			

OpenCL/CUDA futási idők

	Elméleti telj. (GF/s)	5x5 konvolúció	
Intel Core i7 960	~105	84 Mpixel/s Intel OpenCL	20 Mpixel/s AMD OpenCL
Qualcomm Adreno 530	~450	322 Mpixel/s OpenCL	
NVIDIA Tegra TX2 GPU	~500	2000 Mpixel/s CUDA	
NVIDIA GP104	~7000	17930 Mpixel/s CUDA	Pointer casting N/A OPENCL
AMD Radeon 7750M	~820	1956 Mpixel/s OpenCL	
Intel HD Graphics 630	~400	1250 Mpixel/s OpenCL	

Heterogén számítási rendszerek

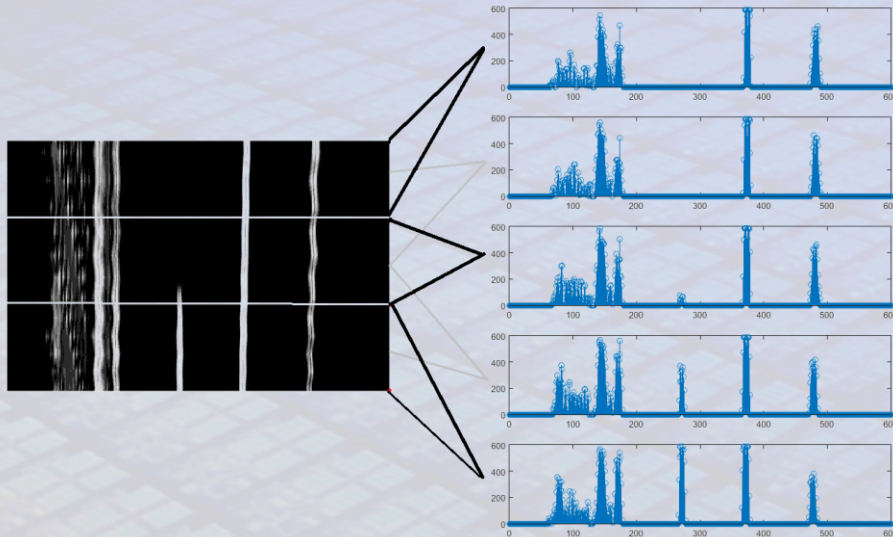
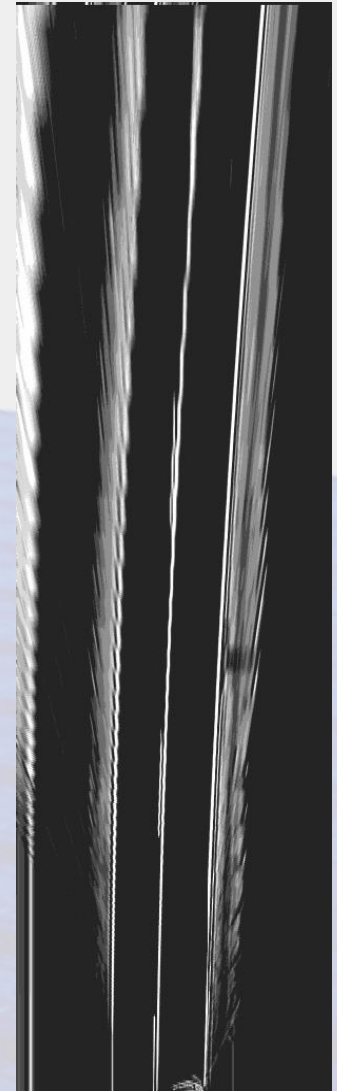
CUDA példa: sávdetektálás

Szántó Péter

BME MIT, FPGA Laboratórium

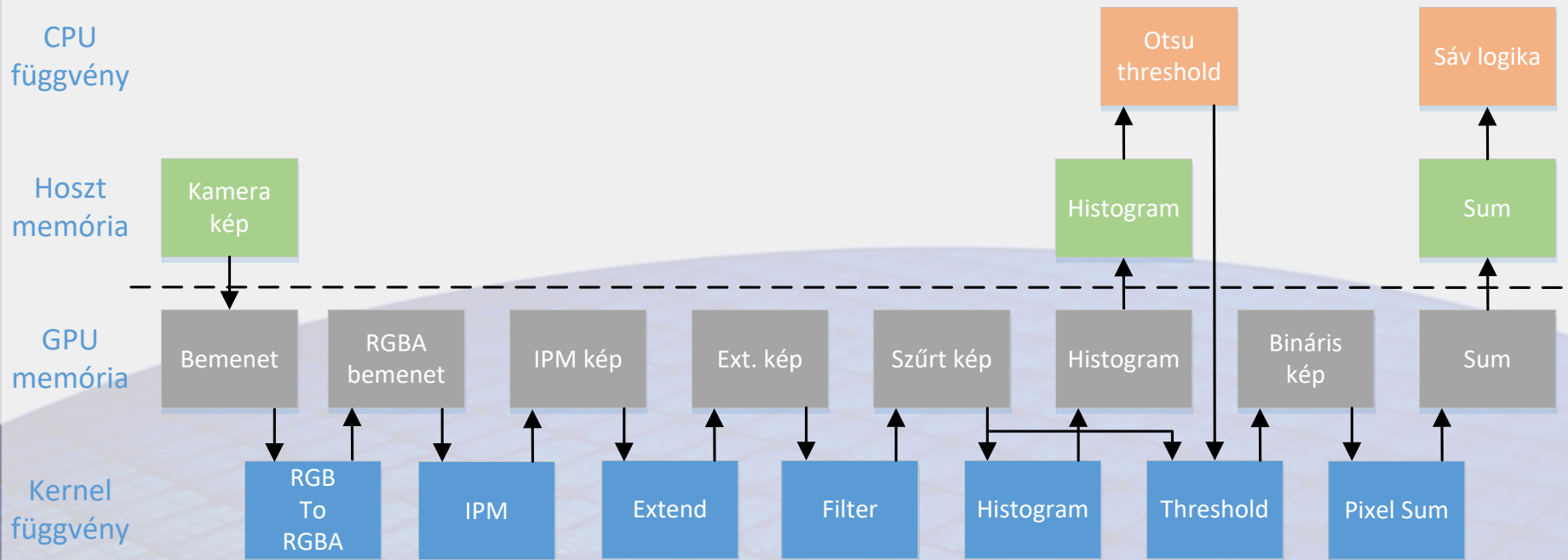
Sávdetektálás

- Bemenet: egyetlen kamera képe
- Kimenet: sáv középvpvonal
- Real-time működés: ~20 ms/kép (50 FPS)



Teljes algoritmus

■ GPU gyorsított rész:

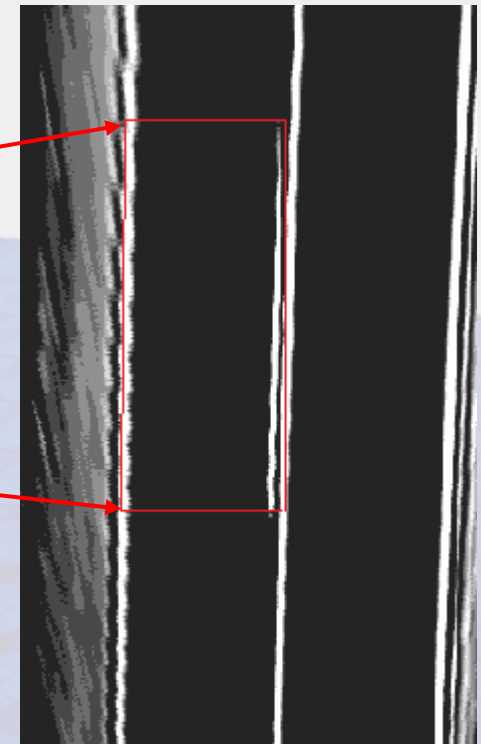


IPM

- **IPM: Inverse Perspective Mapping**

$$r(x) = \frac{m-1}{2} \left(1 + \frac{h - x * \tan \varnothing}{h * \tan \varnothing + x} \cot \alpha_v \right) + 1$$

$$c(x, y) = \frac{n-1}{2} \left(1 + \frac{h - x * \tan \varnothing}{h * \sin \varnothing + x * \cos \varnothing} \cot \alpha_u \right) + 1$$



- **Adott kimeneti területhez tartozó bemeneti terület mérete és formája változó, de lokális 2D terület → textúra**

Részletek...

- RGB → RGBA textúra: bemenet globális memória, kimenet surface („írható textúra”)
- IPM: címszámlálás az előző képletek alapján, textúra olvasás szűréssel
- Extend: bal és jobb alsó sarok kiterjesztése (nagyon nem ideális!)
- Filter: 47 tap-es 1D konvolúció
- Histogram: következő slide
- Threshold: per pixel ~binarizálás
- Pixel Sum: 46 rész-terület oszlop összege

Hisztogram

- **Párhuzamosítás: a kép felosztása kis területekre**
 - Itt rész-hisztogramok számítása
 - Majd a rész-hisztogramok összegzése
- **Minden számítás atomi művelet: egy szál a saját néhány pixelének adatával frissíti a hisztogramot**
 - >Maxwell: rész-hisztogram shared memóriában
 - Kepler: L2 cache-ben (globális memória)
- **Végleges hisztogram: atomi művelet a globális memóriában**
 - Minden szál 1-1 rész-hisztogram értékkel frissíti a végleges hisztogramot

Futási idők: GP104

- H/D Memcopy Σ : ~2,58 ms / frame, kernel Σ : 0,26 ms**

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	216,122 ms	229,22 μ s	n/a	n/a	n/a	n/a	n/a	1,398 MB	6,098 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	216,364 ms	16,417 μ s	[80,23,1]	[16,16,1]	8	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int, int, ...)	216,384 ms	25,248 μ s	[36,38,1]	[16,16,1]	16	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	216,767 ms	151,458 μ s	[5,1,1]	[128,1,1]	12	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned...	217,245 ms	27,777 μ s	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
Memcpy DtoH [sync]	217,275 ms	210,563 μ s	n/a	n/a	n/a	n/a	n/a	1,35 MB	6,411 GB/s
Memcpy HtoD [sync]	218,139 ms	1,28 μ s	n/a	n/a	n/a	n/a	n/a	4,096 kB	3,2 GB/s
kernel_ipm_histogram_sh_int(float*, unsigned char*, in...	218,144 ms	10,528 μ s	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
Memcpy DtoH [sync]	218,157 ms	608 ns	n/a	n/a	n/a	n/a	n/a	4,096 kB	6,737 GB/s
kernel_ipm_threshold(unsigned char*, unsigned char*, ...)	218,533 ms	25,473 μ s	[5,141,1]	[32,16,1]	24	0	0	n/a	n/a
Memcpy DtoH [sync]	218,561 ms	211,971 μ s	n/a	n/a	n/a	n/a	n/a	1,35 MB	6,369 GB/s
kernel_ipm_pixsum(unsigned char*, float*, int, int)	219,351 ms	3,008 μ s	[46,1,1]	[600,1,1]	8	0	0	n/a	n/a
Memcpy DtoH [sync]	219,356 ms	16,864 μ s	n/a	n/a	n/a	n/a	n/a	110,4 kB	6,546 GB/s

- Zerocopy Σ : ~8,82 ms / frame**

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	206,945 ms	228,708 μ s	n/a	n/a	n/a	n/a	n/a	1,398 MB	6,112 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	207,188 ms	18,016 μ s	[80,23,1]	[16,16,1]	8	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int, int, ...)	207,21 ms	25,313 μ s	[36,38,1]	[16,16,1]	16	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	207,465 ms	157,41 μ s	[5,1,1]	[128,1,1]	12	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned...	207,833 ms	2,352 ms	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
kernel_ipm_histogram_sh_int(float*, unsigned char*, in...	210,361 ms	1,033 ms	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
kernel_ipm_threshold(unsigned char*, unsigned char*, ...)	211,866 ms	2,453 ms	[5,141,1]	[32,16,1]	24	0	0	n/a	n/a
kernel_ipm_pixsum(unsigned char*, float*, int, int)	214,585 ms	146,947 μ s	[46,1,1]	[600,1,1]	8	0	0	n/a	n/a

Futási idők: Tegra X1

■ H/D Memcopy Σ : ~5.87 ms / frame

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	208.102 ms	161.772 μ s	n/a	n/a	n/a	n/a	n/a	3 MB	8.64 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	208.394 ms	202.658 μ s	[80,23,1]	[16,16,1]	8	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int)	208.598 ms	487.14 μ s	[36,38,1]	[16,16,1]	19	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	209.347 ms	693.287 μ s	[5,1,1]	[128,1,1]	12	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned char*, int, int)	210.189 ms	546.047 μ s	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
Memcpy DtoH [sync]	210.736 ms	135.99 μ s	n/a	n/a	n/a	n/a	n/a	5 MB	9.927 GB/s
Memcpy HtoD [sync]	211.416 ms	677 ns	n/a	n/a	n/a	n/a	n/a	16 kB	6.05 GB/s
kernel_ipm_histogram_sh_int(float*, unsigned char*, int, int)	211.515 ms	419.952 μ s	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
Memcpy DtoH [sync]	211.936 ms	1.562 μ s	n/a	n/a	n/a	n/a	n/a	16 kB	2.622 GB/s
kernel_ipm_threshold(unsigned char*, unsigned char*, int, int)	212.96 ms	277.19 μ s	[5,141,1]	[32,16,1]	24	0	0	n/a	n/a
Memcpy DtoH [sync]	213.238 ms	132.45 μ s	n/a	n/a	n/a	n/a	n/a	5 MB	10.193 GB/s
kernel_ipm_pixsum(unsigned char*, float*, int, int)	213.971 ms	19.376 μ s	[46,1,1]	[600,1,1]	8	0	0	n/a	n/a
Memcpy DtoH [sync]	214.016 ms	8.958 μ s	n/a	n/a	n/a	n/a	n/a	4 kB	12.324 GB/s

■ Zero copy Σ : ~5,6 ms / frame

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	162.259 ms	278.077 μ s	n/a	n/a	n/a	n/a	n/a	1.398 MB	5.027 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	162.951 ms	202.191 μ s	[80,23,1]	[16,16,1]	8	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int)	163.158 ms	485.996 μ s	[36,38,1]	[16,16,1]	19	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	163.906 ms	678.187 μ s	[5,1,1]	[128,1,1]	12	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned char*, int, int)	164.784 ms	546.779 μ s	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
kernel_ipm_histogram_sh_int(float*, unsigned char*, int, int)	165.675 ms	419.85 μ s	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
kernel_ipm_threshold(unsigned char*, unsigned char*, int, int)	167.457 ms	277.452 μ s	[5,141,1]	[32,16,1]	24	0	0	n/a	n/a
kernel_ipm_pixsum(unsigned char*, float*, int, int)	167.894 ms	19.792 μ s	[46,1,1]	[600,1,1]	8	0	0	n/a	n/a

■ Tegra X1: Σ : 5,6 ms, kernel Σ : 2,9 ms

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	162.259 ms	278.077 μ s	n/a	n/a	n/a	n/a	n/a	1.398 MB	5.027 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	162.951 ms	202.191 μ s	[80,23,1]	[16,16,1]	8	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int)	163.158 ms	485.996 μ s	[36,38,1]	[16,16,1]	19	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	163.906 ms	678.187 μ s	[5,1,1]	[128,1,1]	12	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned char*, int, int)	164.784 ms	546.779 μ s	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
kernel_ipm_histogram_sh_int(float*, unsigned char*, int, int)	165.675 ms	419.85 μ s	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
kernel_ipm_threshold(unsigned char*, unsigned char*, int, int)	167.457 ms	277.452 μ s	[5,141,1]	[32,16,1]	24	0	0	n/a	n/a
kernel_ipm_pixsum(unsigned char*, float*, int, int)	167.894 ms	19.792 μ s	[46,1,1]	[600,1,1]	8	0	0	n/a	n/a

■ AGX Xavier: Σ : 4,23ms, kernel Σ : 1,87 ms

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	235.19839 ms	168.302 μ s	n/a	n/a	n/a	n/a	n/a	3 MB	8.305 GB/s
kernel_ipm_3to4(unsigned char*, int, int)	235.49227 ms	146.331 μ s	[80,23,1]	[16,16,1]	16	0	0	n/a	n/a
kernel_ipm_tex(float, float, unsigned char*, int, int)	235.64555 ms	235.943 μ s	[36,38,1]	[16,16,1]	21	0	0	n/a	n/a
kernel_ipm_extend(unsigned char*, int, int, int)	236.18453 ms	718.878 μ s	[5,1,1]	[128,1,1]	20	0	0	n/a	n/a
kernel_ipm_filter_7_32x16_5p(unsigned char*, unsigned char*, int, int)	237.10963 ms	230.402 μ s	[5,36,1]	[32,16,1]	32	16896	0	n/a	n/a
kernel_ipm_histogram_sh_int(float*, unsigned char*, int, int)	237.60406 ms	91.085 μ s	[2,50,1]	[256,1,1]	32	1024	0	n/a	n/a
kernel_ipm_threshold(unsigned char*, unsigned char*, int, int)	238.79076 ms	434.735 μ s	[5,141,1]	[32,16,1]	27	0	0	n/a	n/a
kernel_ipm_pixsum(unsigned char*, float*, int, int)	239.42585 ms	15.694 μ s	[46,1,1]	[600,1,1]	16	0	0	n/a	n/a