# Újrakonfigurálható technológiák nagy teljesítményű alkalmazásai

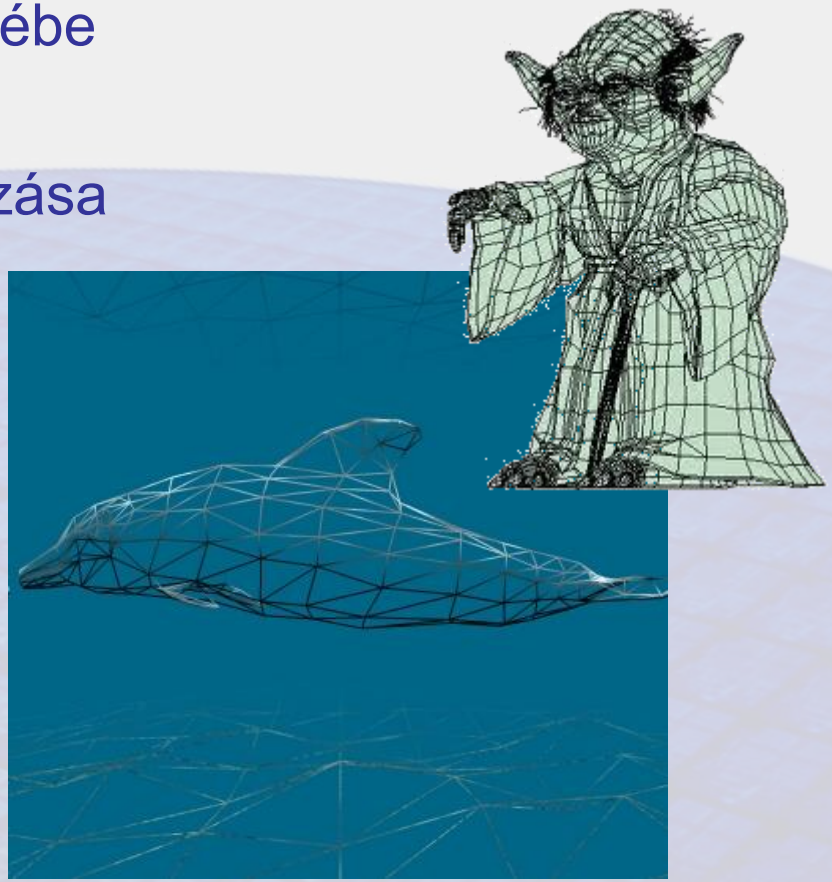
## GPU-k, GPGPU – CUDA

**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

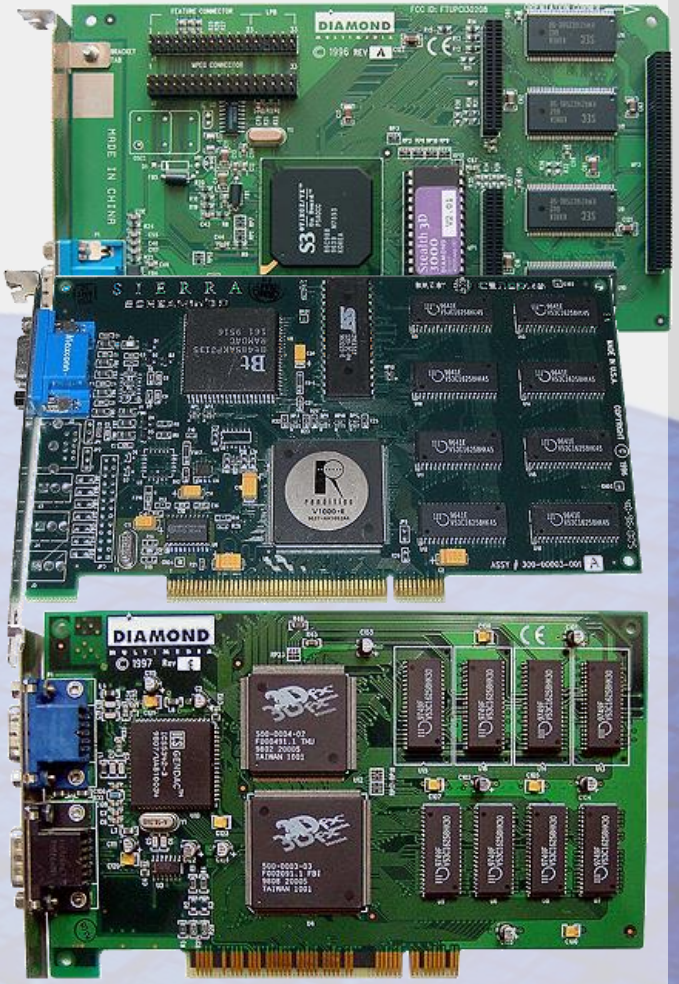
- **Színinformáció 4 csatornán: RGBA**
- **Tipikusan SP lebegőpontos számítás csatornánként**
- **Háromszög csúcspontok színének lineáris interpolációja**
- **Textúra leképzés**
  - „kép ráfeszítése” az objektum háromszögre
  - lokális, jól cache-elhető hozzáférés
  - bilineáris szűrés dedikált HW-rel





# GPU-k fejlődése (1)

- **1995: S3 Virge, ATI Rage 3D/Rage II (CPU: 486DX100, Pentium Pro)**
  - csak fix funkciós raszterizáció
  - 66 MHz, 4 MB RAM
  - „grafikus lassító”
- **1996 (AMD K5)**
  - 3Dfx Voodoo
    - csak 3D
    - 50 MHz, 4 MB
    - Saját API: Glide
  - Rendition Verite V1000
    - 25 MHz, 4 MB
    - DSP jellegű (?)



# GPU-k fejlődése (2)

- **1997 (Pentium MMX, Pentium 2, K6)**
  - NVIDIA Riva128
  - Rendition V2000
  - 3Dfx Voodoo Rush
- **1998**
  - S3 Savage3D
  - Matrox G100
  - Intel i740
  - 3Dfx Voodoo2
  - NVIDIA Riva TNT
  - ATI Rage 128



# GPU-k fejlődése (3)

- **1999: HW T&L (Pentium 3, Athlon)**
  - PowerVR Kyro, 3Dfx Voodoo3
  - HW T&L: S3 Savage 2000, NVIDIA Geforce 256
- **2000 (Pentium 4, Athlon XP)**
  - 3Dfx Voodoo4/Voodoo5
  - NVIDIA Geforce 2,
    ATI Radeon 7200
- **2001: Programozható shader-ek**
  - NVIDIA Geforce 3
  - ATI Radeon 8500
- **2002: Lebegőpontos feldolgozók (vertex & pixel shader)**
  - ATI Radeon 9700
  - NVIDIA GeForce FX

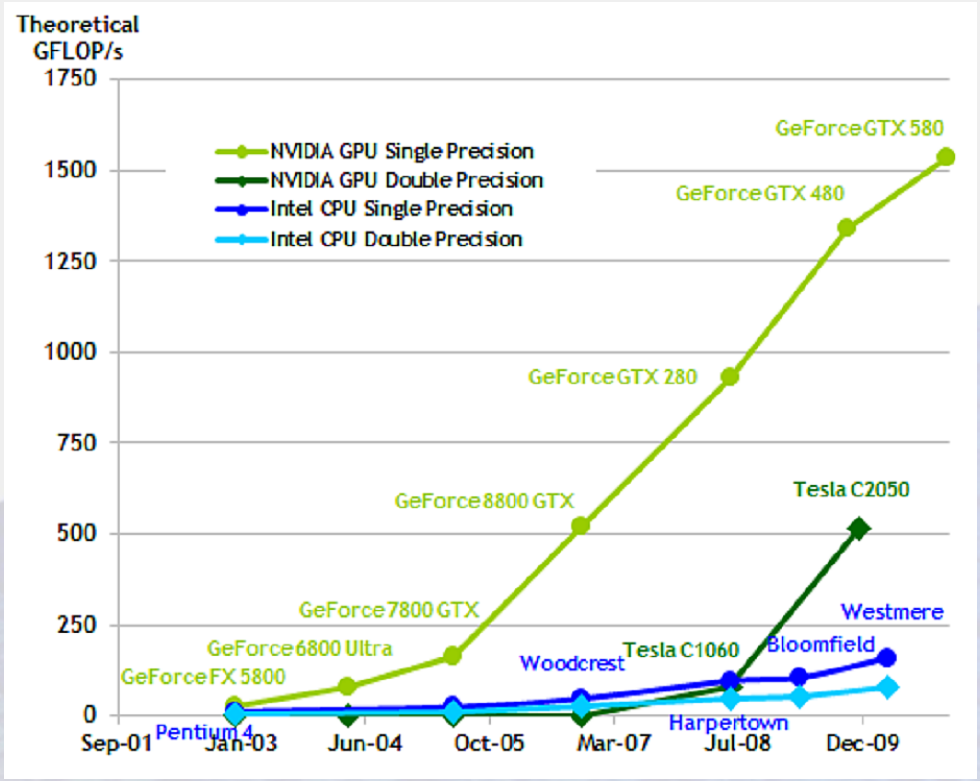


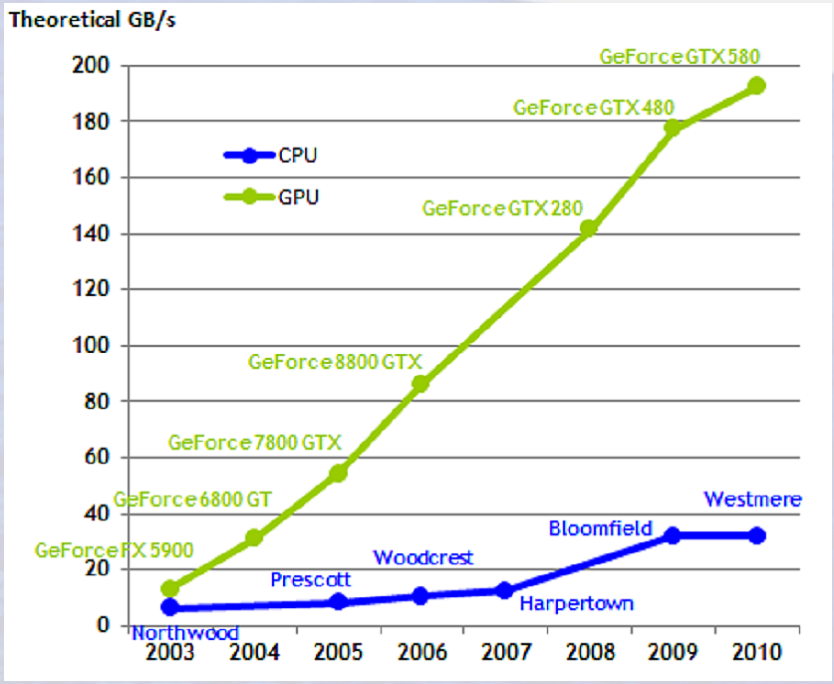
# GPU-k fejlődése (4)

- **2003 – 2007 (Athlon 64)**
  - Radeon 9800, X800, GeForce 6
  - Radeon X1k, GeForce 7
- **2006: „Unified Shader”**
  - NVIDIA G80 (GeForce 8)
    - CUDA képes!
  - ATI Radeon 2000
- **2008**
  - ATI HD 4000
  - NVIDIA G200
- **2009**
  - ATI HD 5000



# CPU – GPU elméleti sebességek



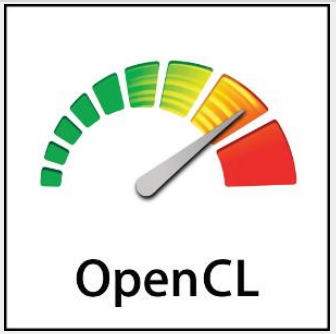
# GPU programozás (1)

- **3D megjelenítéshez kapcsolódó shader-ek**
  - Microsoft Direct3D HLSL
  - OpenGL Shading Language (GLSL)
- **ATI Compute Abstraction Layer (CAL)**
  - RIP
- **NVIDIA CUDA**
  - C, Fortran



# GPU programozás (2)

- **OpenCL**
  - Platfrom és gyártó független
  - CPU, GPU, Cell, ….




- **Microsoft DirectCompute**
  - Gyártó független, de DirectX-hez kötött

- **Microsoft C++ AMP (Accelerated Massive Parallelism)**
  - Adatpárhuzamos feldolgozási könyvtár (Windows 7)
  - C++ nyelv
  - DirectCompute-ra fordul
  - (Linux támogatás OpenCL lesz, ha lesz)

# 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**

# GPGPU chipek

- **ATI**
  - Radeon HD HD 4xxx
    - CAL, OpenCL (lassú!), DX-CS 4.1
  - Radeon HD 5xxx, 6xxx széria
    - CAL, OpenCL, DX-CS 5.0
  - Radeon HD 7xxx, 8xxx szeria
    - OpenCL, DX-CS 5.0
- **NVIDIA**
  - G80: CUDA 1.0, OpenCL, DX-CS 4.0
  - G9x: CUDA 1.1, OpenCL, DX-CS 4.0
  - G21x: CUDA 1.2, OpenCL, DX-CS 4.1
  - G200: CUDA 1.3, OpenCL, DX-CS 4.0
  - Fermi: CUDA 2.x, OpenCL, DX-CS 5.0
  - Kepler: CUDA 3.x, OpenCL, DX-CS 5.0






# OpenCL

- **Eszköz és operációs rendszer független**
- **AMD APP SDK**
  - OpenCL 1.2 driver
    - CPU
    - AMD GPU
  - AMD CodeXL debugger: kernel debug GPU-n
- **Intel OpenCL SDK**
  - OpenCL 1.2 driver CPU-hoz
  - OpenCL 1.1 driver HD Graphics 4000/2500-hoz
  - Kernel debug
- **NVIDIA**
  - OpenCL 1.1 driver

# GPGPU mobil eszközökben

- **OpenCL 1.0 Embedded Profile**
  - Enyhébb követelmények (pl. számítási precizitás)
- **Operációs rendszer támogatás (még nem publikus)**
  - iOS
  - Android
- **Kompatibilis GPU-k**
  - ARM Mali T6xx
  - Imagination Technologies PowerVR Series 5 (SGX)
  - Imagination Technologies PowerVR Series 6
  - Qualcomm Adreno 3xx
  - Vivante GC800

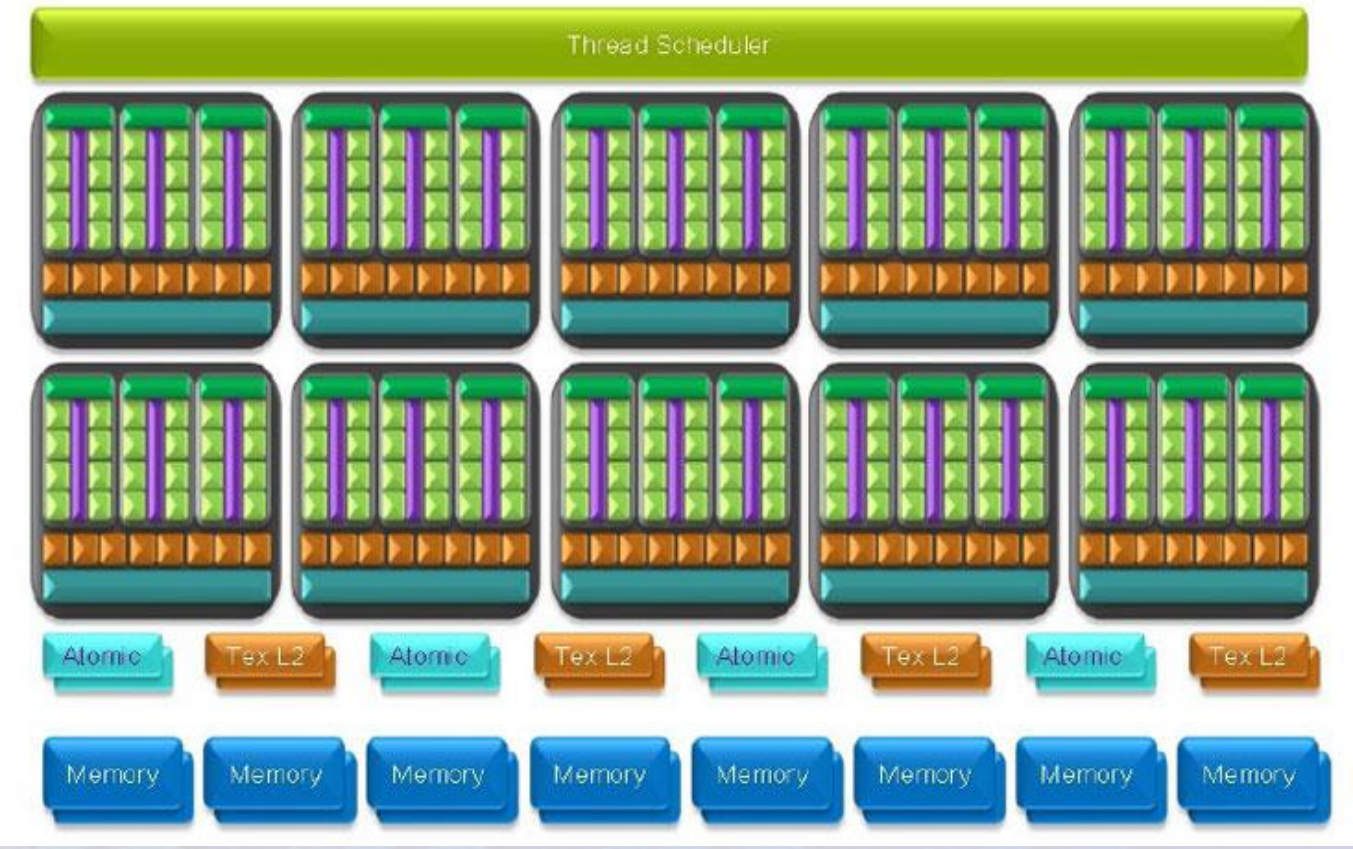
# NVIDIA GTX 280

- **NVIDIA G200 chip**
  - 240 ALU
  - 1,4 milliárd tranzisztor
  - 1300 MHz
  - 936 GFLOPS SP MAD
  - 80 GFLOPS DP MAD
  - 312 GOPS INT
- **1 Gbyte DDR3 memória**
  - 512 bites busz
  - 2200 MHz
  - 140,8 Gbyte/s sávszélesség
- **~230 W fogyasztás**



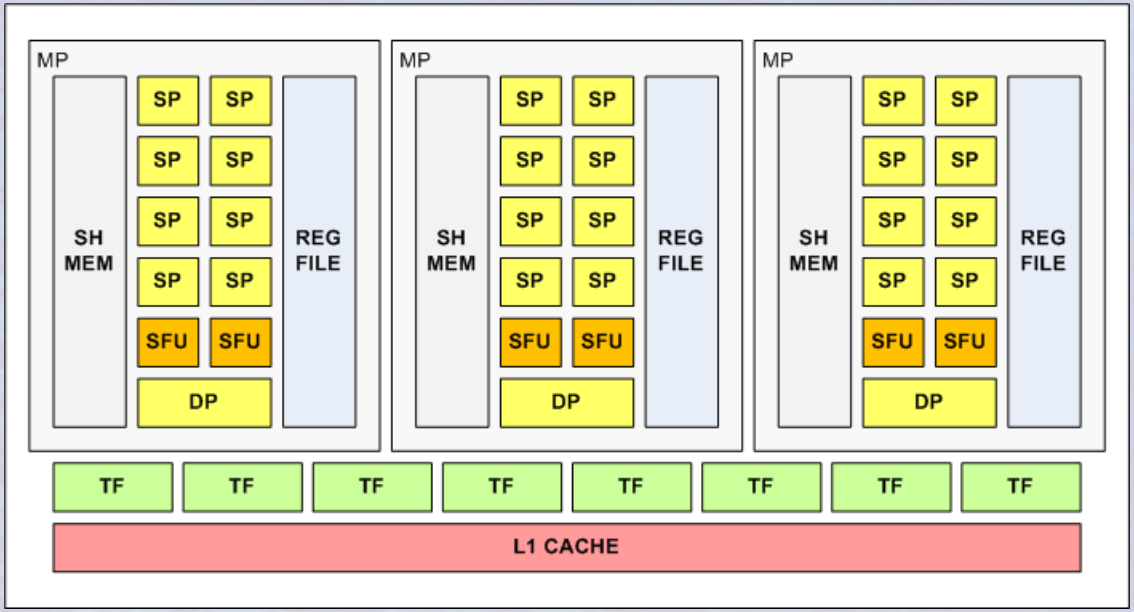
# NVIDIA G200

- **10 Thread Processing Block (TPC)**
- **3 Streaming Multiprocessor (SM) / TPC**
- **SM:**
  - 8 SP
  - 2 SFU
  - 1 DP
  - Shared Memory



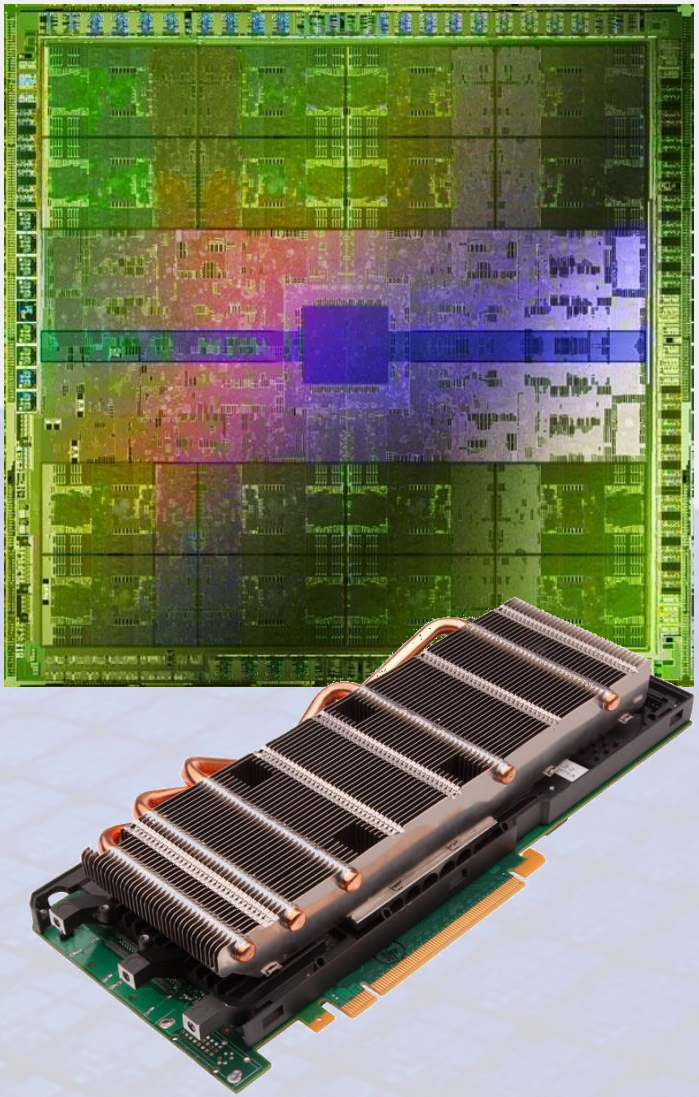
# NVIDIA G200 TPC

- **SP: skalár „thread processor” (ALU)**
  - FP & INT műveletek (INT szorzás: 24 bit)
- **DP: dupla pontosságú lebegőpontos egység**
  - DP & 32 bit INT MUL
- **SFU: transzcendens operátorok**
- **SH MEM: 16 kbyte osztott memória**
  - 16 bank
- **16k regiszter**



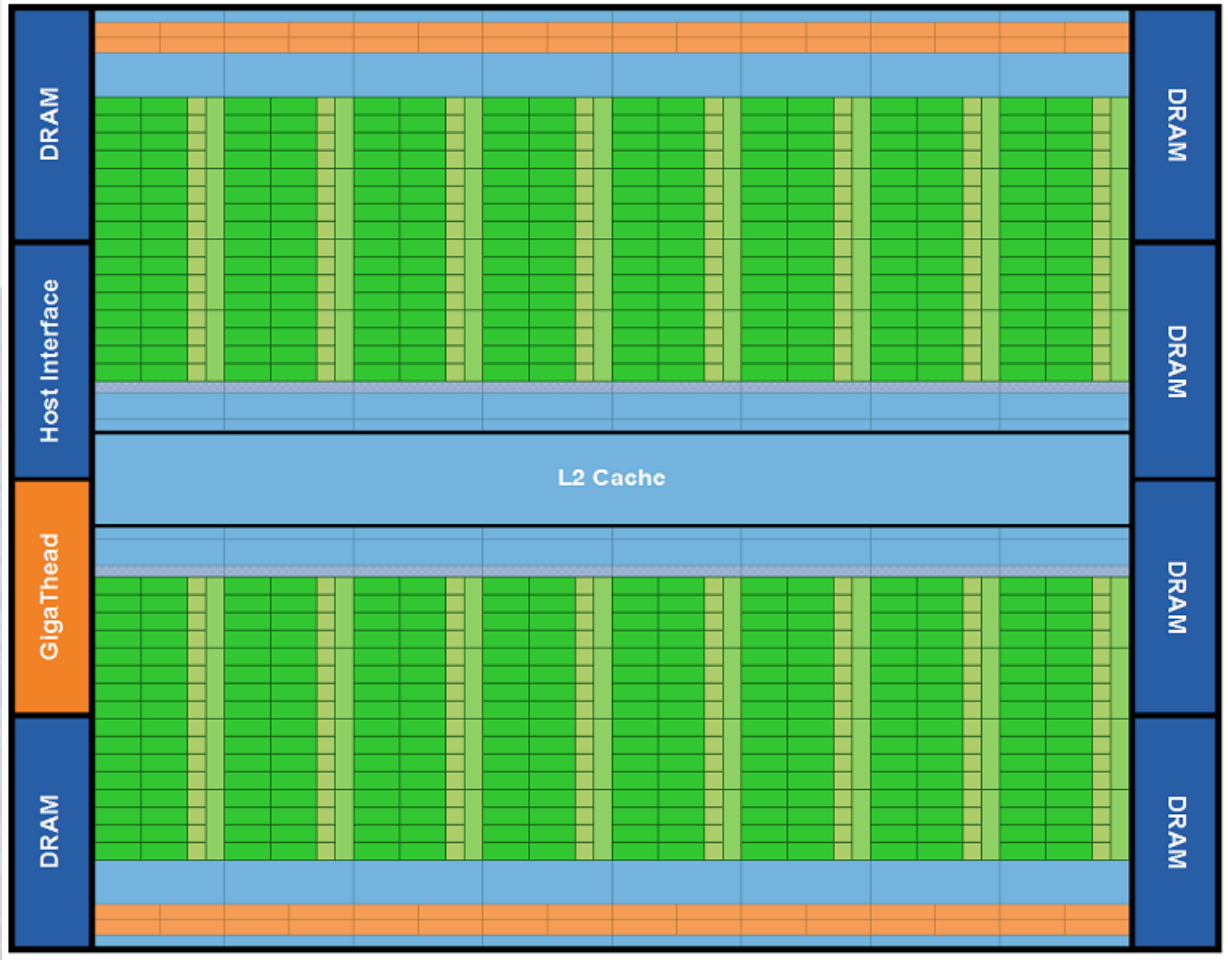
# NVIDIA GF100 (Fermi)

- **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**



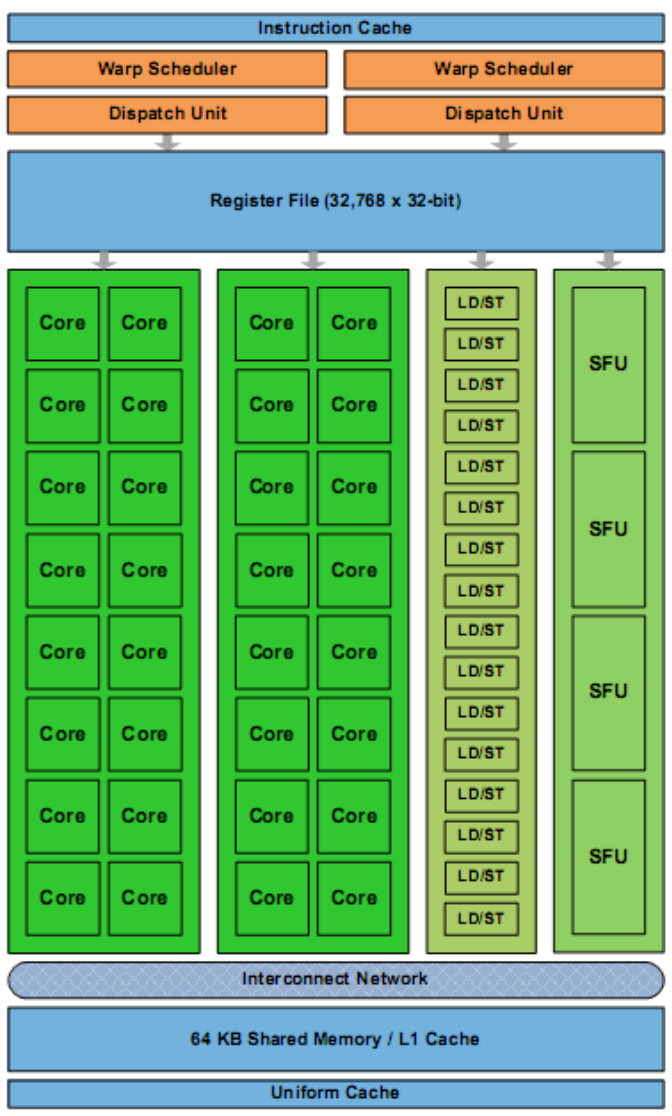
# NVIDIA GF100 architektúra

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



# NVIDIA GF100 SM

| | GF100 | G200 |
|---|---|---|
| **ALU** | 32 | 8 |
| **SFU** | 2 | 4 |
| **Regiszter** | 32k*32 | 16k*32 |
| **Shared Memory** | 16 – 48 kbyte | 16 kbyte |
| **Cache** | 16 – 48 kbyte | ---- |
| **Threads** | 1536 | 1024 |

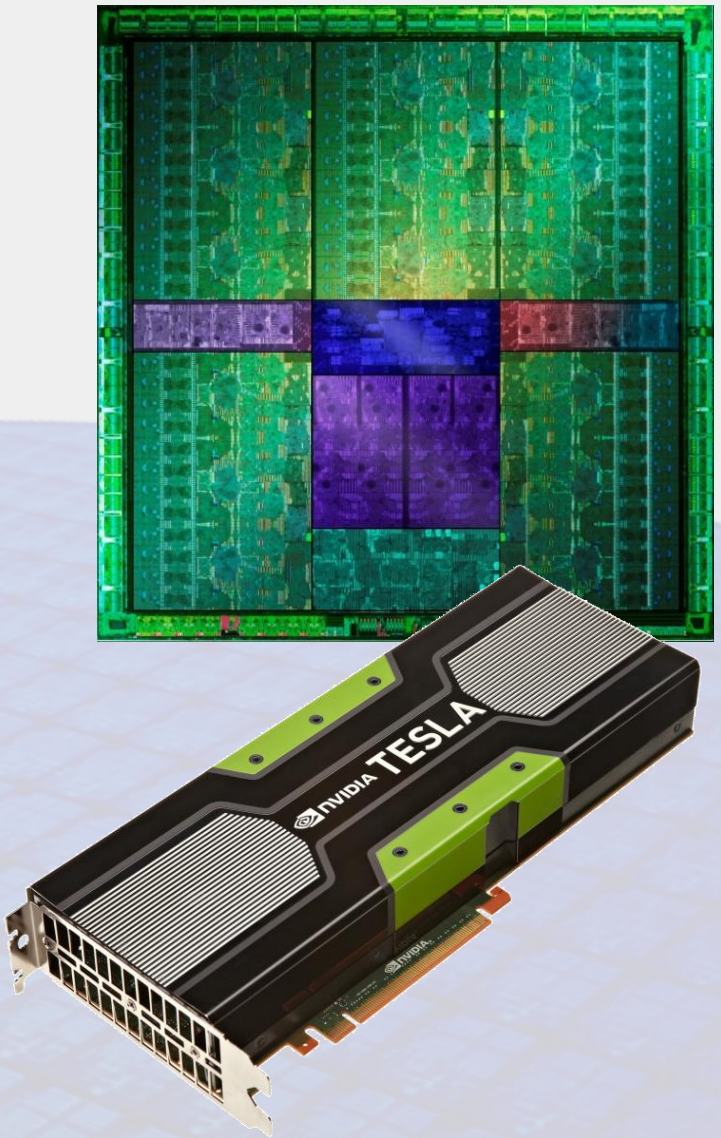


# NVIDIA GF100 SM

- **FPU**
  - Nincs külön 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

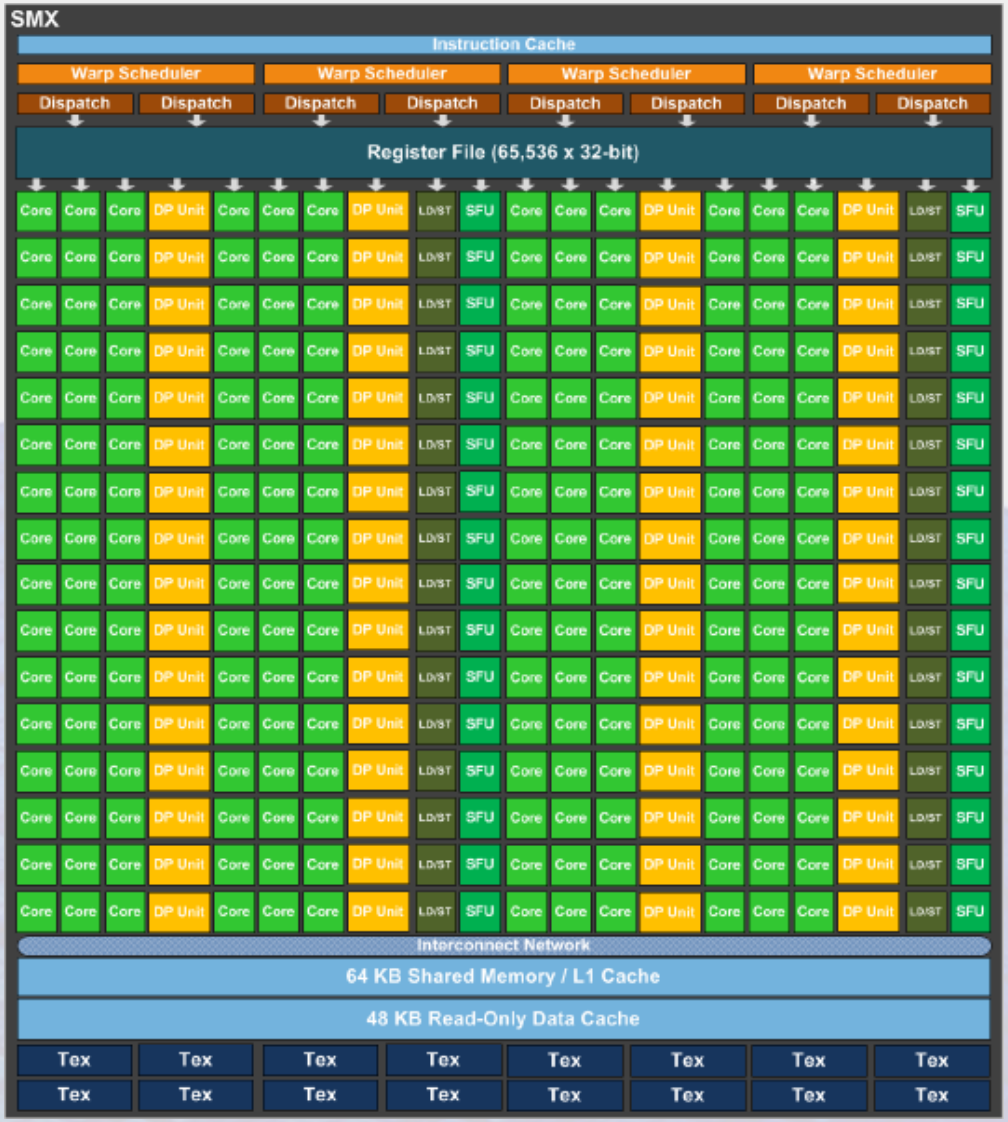
# NVIDIA GK110 (Kepler)

- **NVIDIA GK110 chip (Tesla K20X)**
  - 2688 ALU
  - 7,1 milliárd tranzisztor
  - 732 MHz (?)
  - 3,95 TFLOPS SP MAD
  - 1,31 GFLOPS DP MAD
- **DDR5 memória**
  - 384 bites busz
  - ECC támogatás
  - 5200 MHz
  - 250 Gbyte/s sávszélesség
- **235 W fogyasztás**



# NVIDIA GK110 SMX

- 14 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

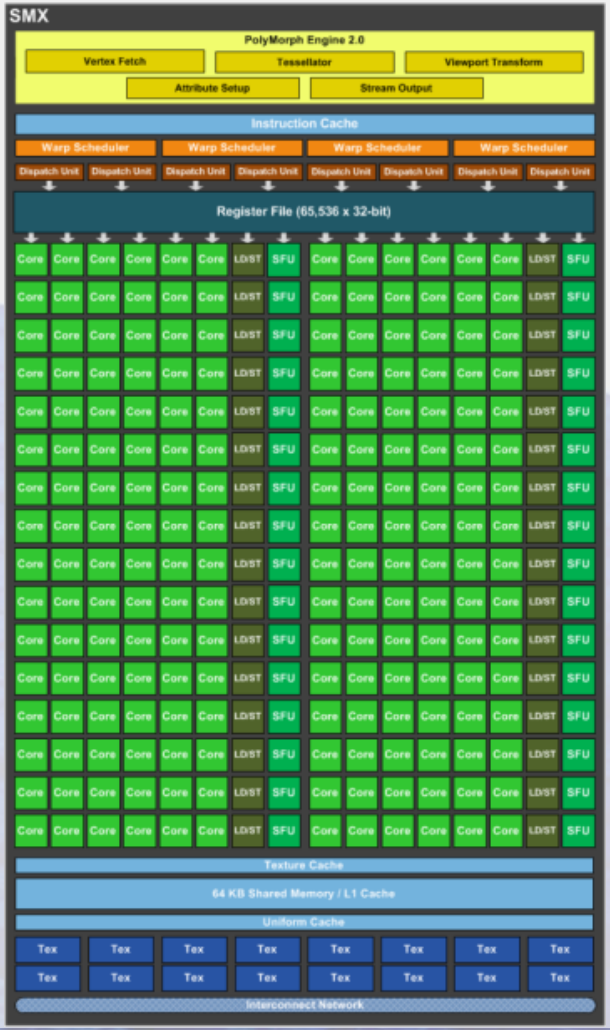


# GK110 új funkciók (1)

- **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

# 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**



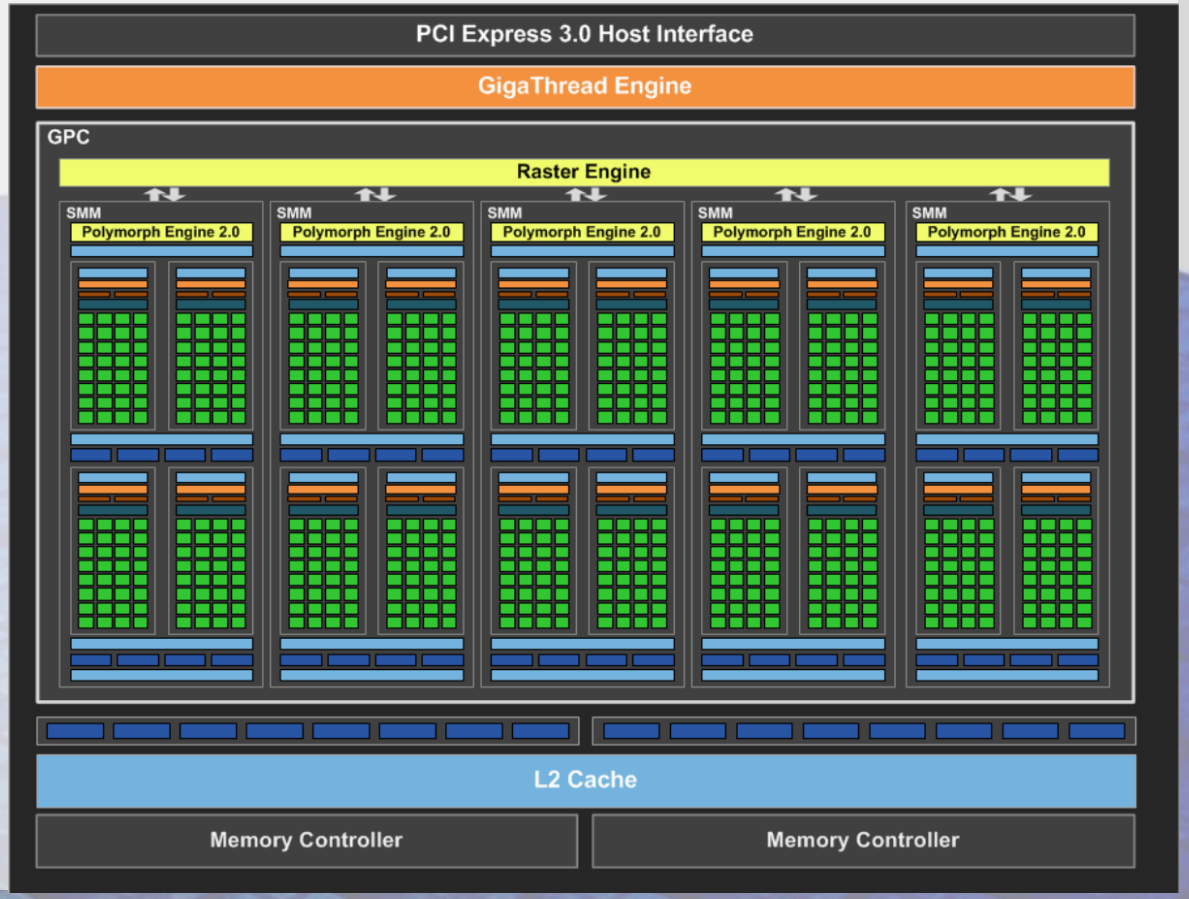
# GK110 új funkciók (2)

- **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 Maxwell (GM107) 1.

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

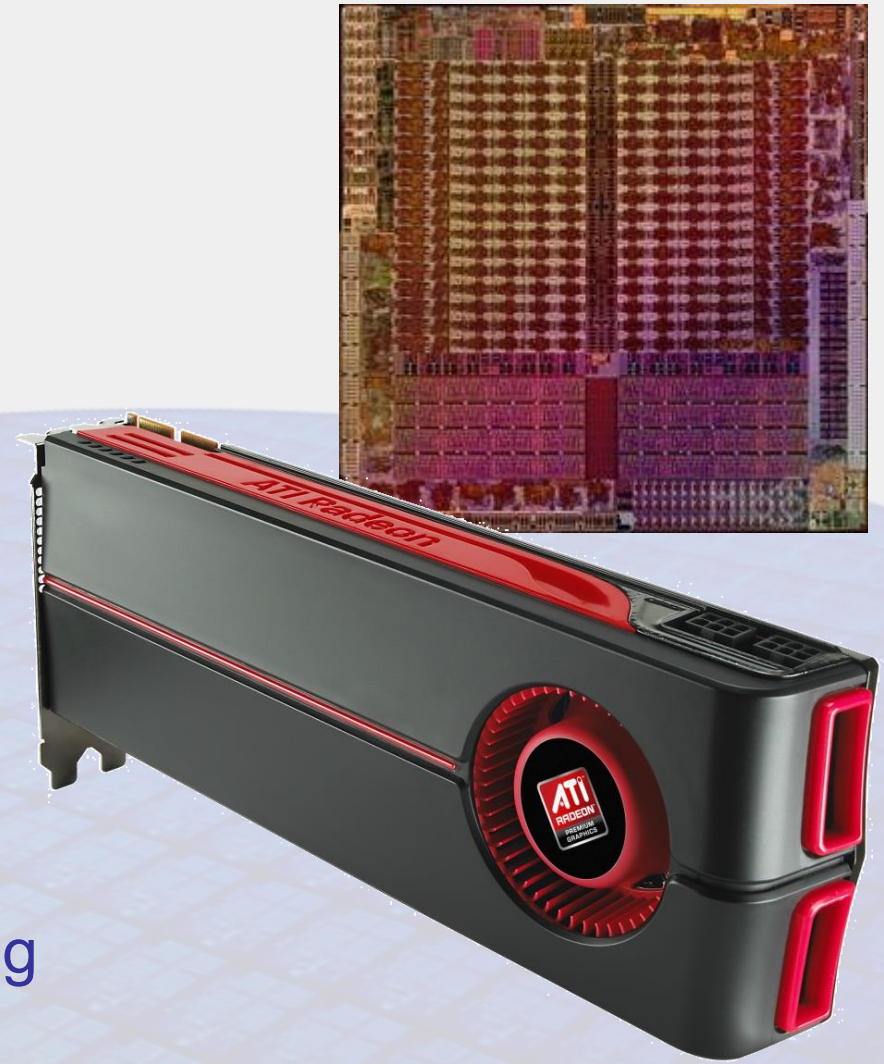


# NVIDIA Maxwell (GM107) 2.

- **GM107: „első generációs Maxwell”: SM = SMM**
  - 32 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
    - 64 KB Shared Memory / SMM
    - Max. 48 KB / thread block → praktikusan 32 KB / tb
  - L2 cache: 2 MB

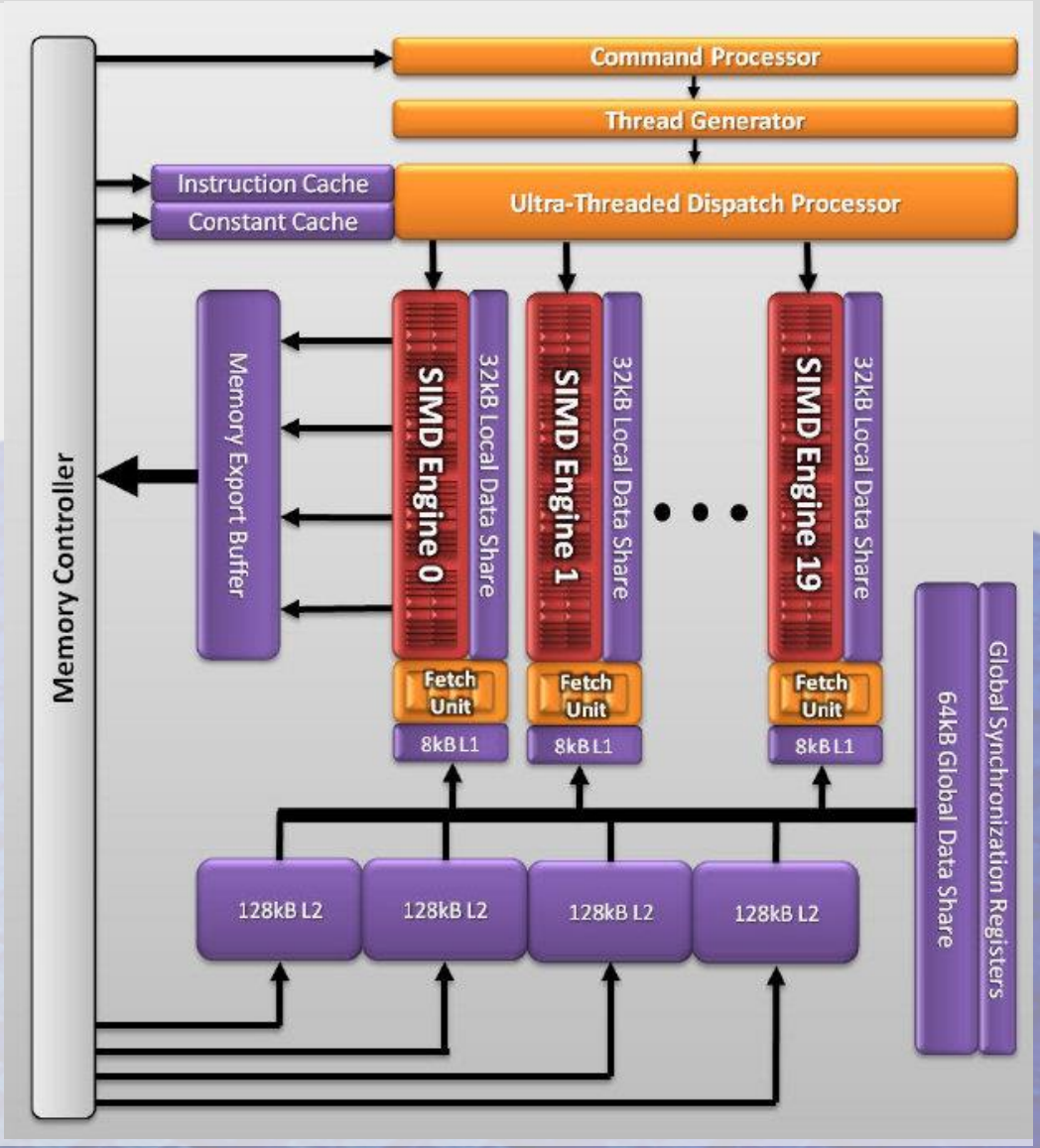
# AMD Radeon HD 5870

- **AMD Cypress chip**
  - 1600 ALU
  - 2,15 milliárd tranzisztor
  - 850 MHz
  - 2700 GFLOPS SP MAD
  - 544 GFLOPS DP MAD
  - 1088 GOPS INT
- **1 Gbyte DDR5 memória**
  - 256 bites busz
  - 4800 MHz
  - 153,6 Gbyte/s sávszélesség
- **~190 W fogyasztás**



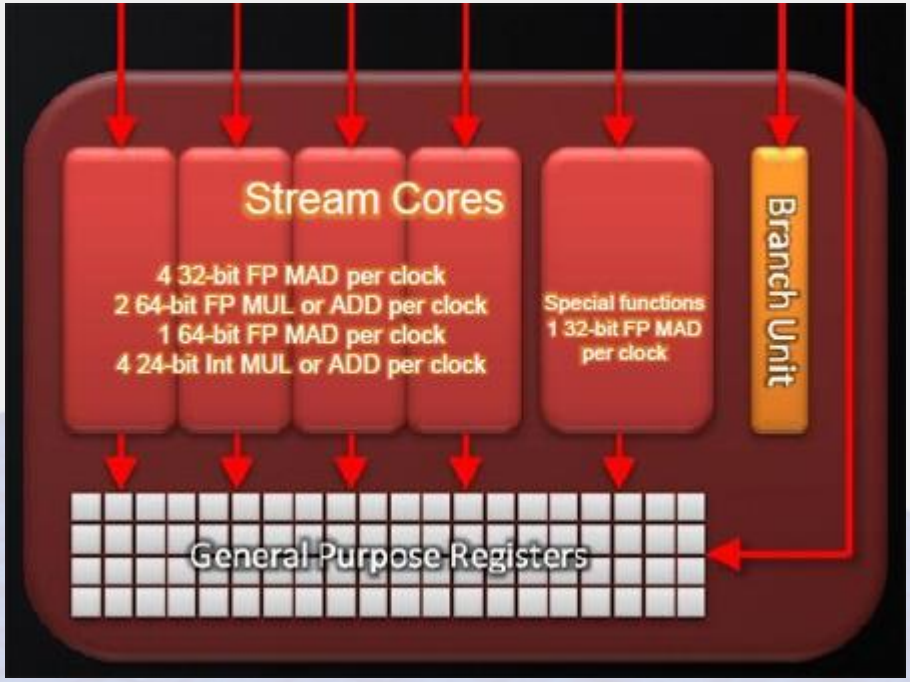
# AMD Cypress architektúra

- **20 SIMD egység**
  - 16 VLIW lane/SIMD
    - 16*5 ALU
  - Statikus ütemezés (fordító)
  - 8 kbyte L1 cache
  - 32 kbyte osztott memória
- **64 kbyte osztott memória**
- **4x128 kbyte L2 cache**



# AMD Cypress VLIW

- **4 SLIM ALU**
  - 1 FP MAD/ADD/MUL
  - 1 INT ADD/AND/CMP
  - 1 24-bit INT MUL
  - Bit számlálás, bit-mező kiválasztás/írás
- **1 SF ALU**
  - FP MAD
  - 32-bit INT MUL
  - transzcendens függvények
- **Regiszter file**
  - 1024 db 4x32 bit regiszter
  - Max. 3 cím/utasítás



# ATI Cypress LDS/GDS

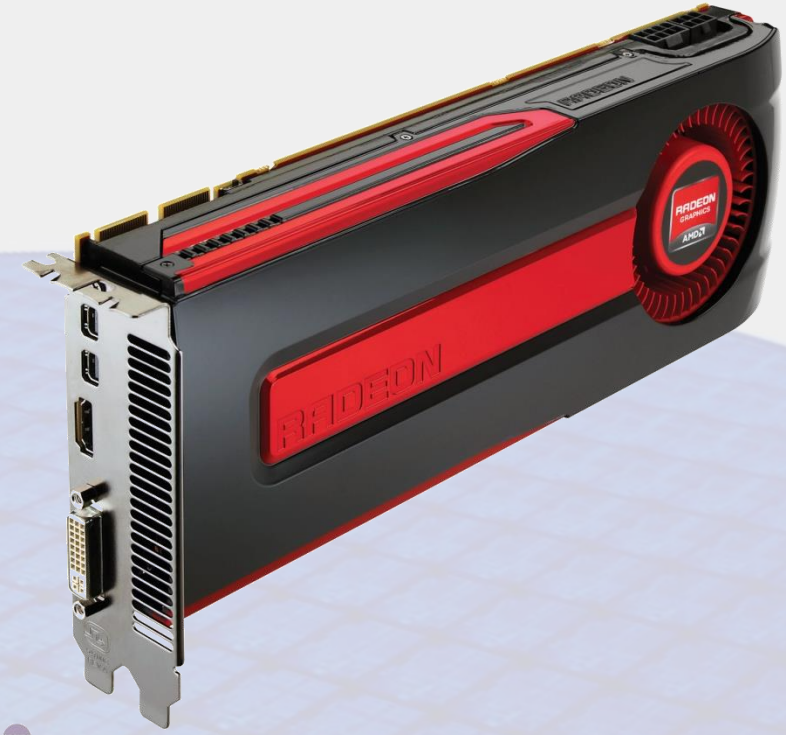
- **LDS: Local Data Share**
  - HD 4xxx sorozatnál nem teljesen OpenCL kompatibilis
- **SIMD egységenként 32 kbyte memória**
  - Azaz 16 VLIW ALU-hoz egy
- **Minden SIMD olvashatja és írhatja**
  - Atomic utasítások
- **32 bank**
  - Párhuzamos hozzáférés; automatikus konfliktus feloldás
  - „Broadcast” olvasás
- **GDS: Global Data Share**
  - 64 kbyte, minden SIMD egység hozzáfér
  - OpenCL kiterjesztéssel használható, nem szabványos

# HD 6xxx/7xxx

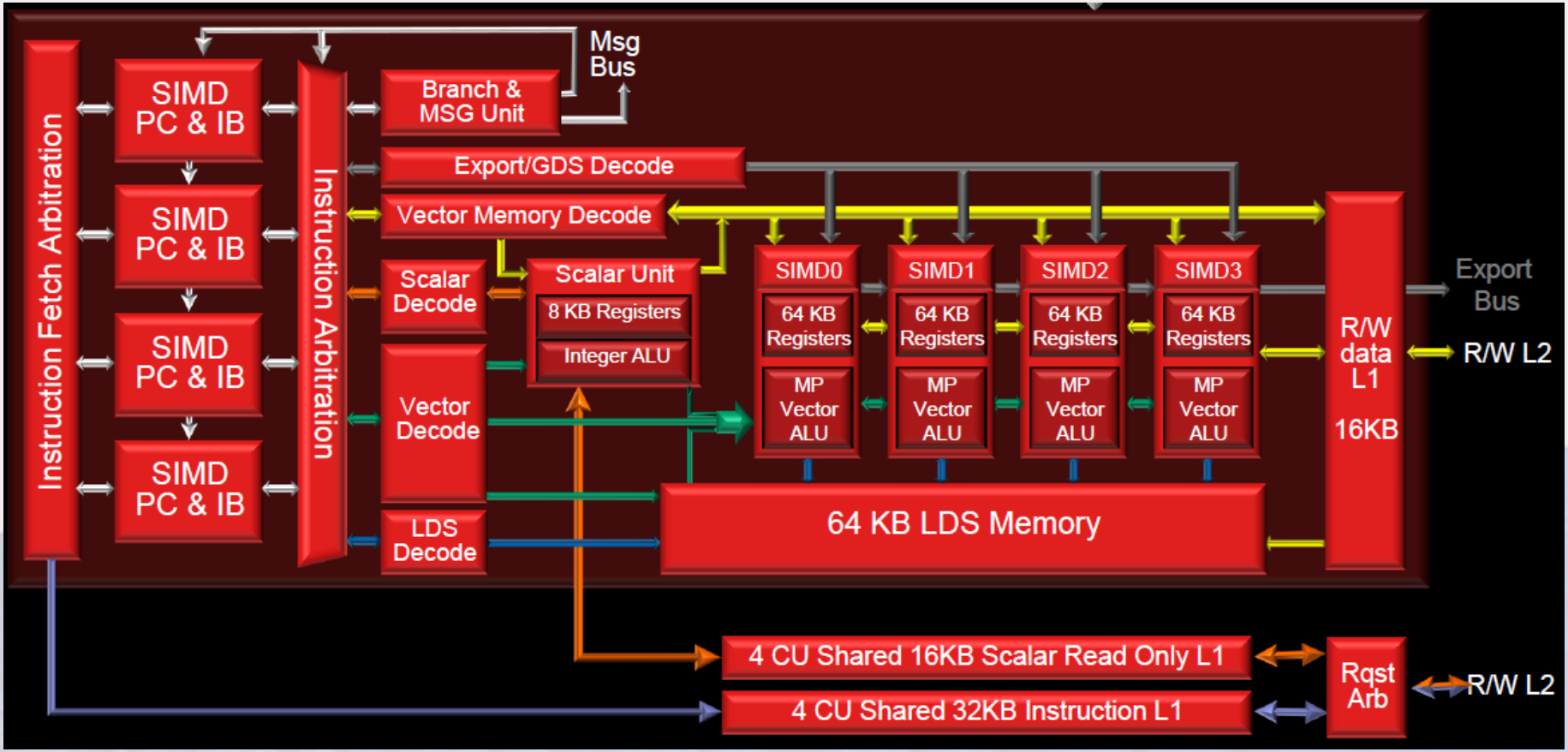
- **Radeon HD 68xx (Barts)**
  - Max. 1120 ALU, nincs architektúra változás
- **Radeon HD 69xx (Cayman)**
  - VLIW5 helyett VLIW4 architektúra
  - Max. 1536 ALU
- **Radeon HD 7xxx (GCN - Graphics Core Next)**
  - Teljesen új architektúra
  - GPGPU igényeknek jobban megfelel

# AMD 7970

- **AMD GCN chip (2011. dec)**
  - 2048 ALU
  - 4,3 milliárd tranzisztor
  - 925 MHz órajel
  - 3788 Gflops/s
  - 947 DP Gflops/s
  - 1900 INT GOPS
- **3 Gbyte DDR5 memória**
  - 384 bites busz
  - 5500 MHz
  - 260 Gbyte/s sávszélesség
- **PCI Express 3.0**



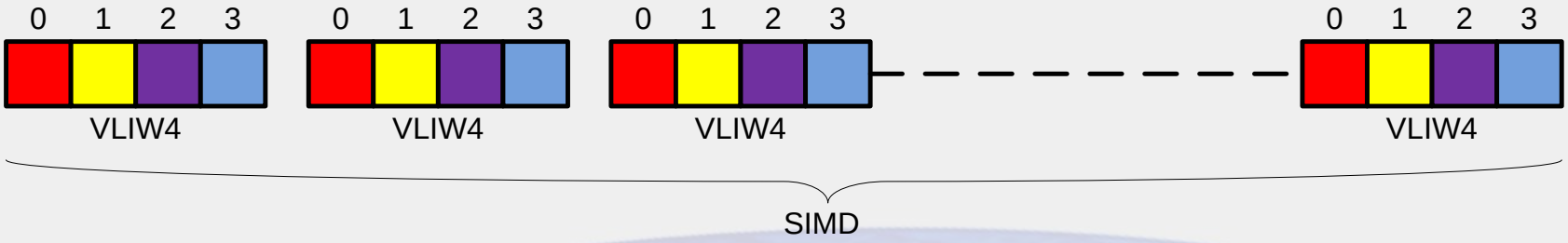
# AMD GCN – Compute Unit



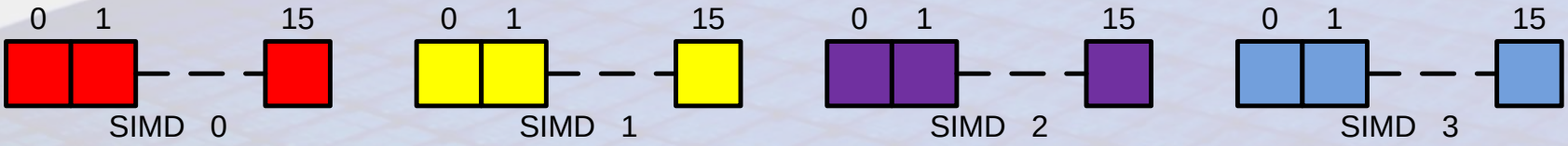

- 4 db 16 széles SIMD egység (=64 ALU)
- Regiszter file, cache, lokális memória (32 bank)

# VLIW4 vs. GCN

## AMD VLIW4 SIMD architektúra

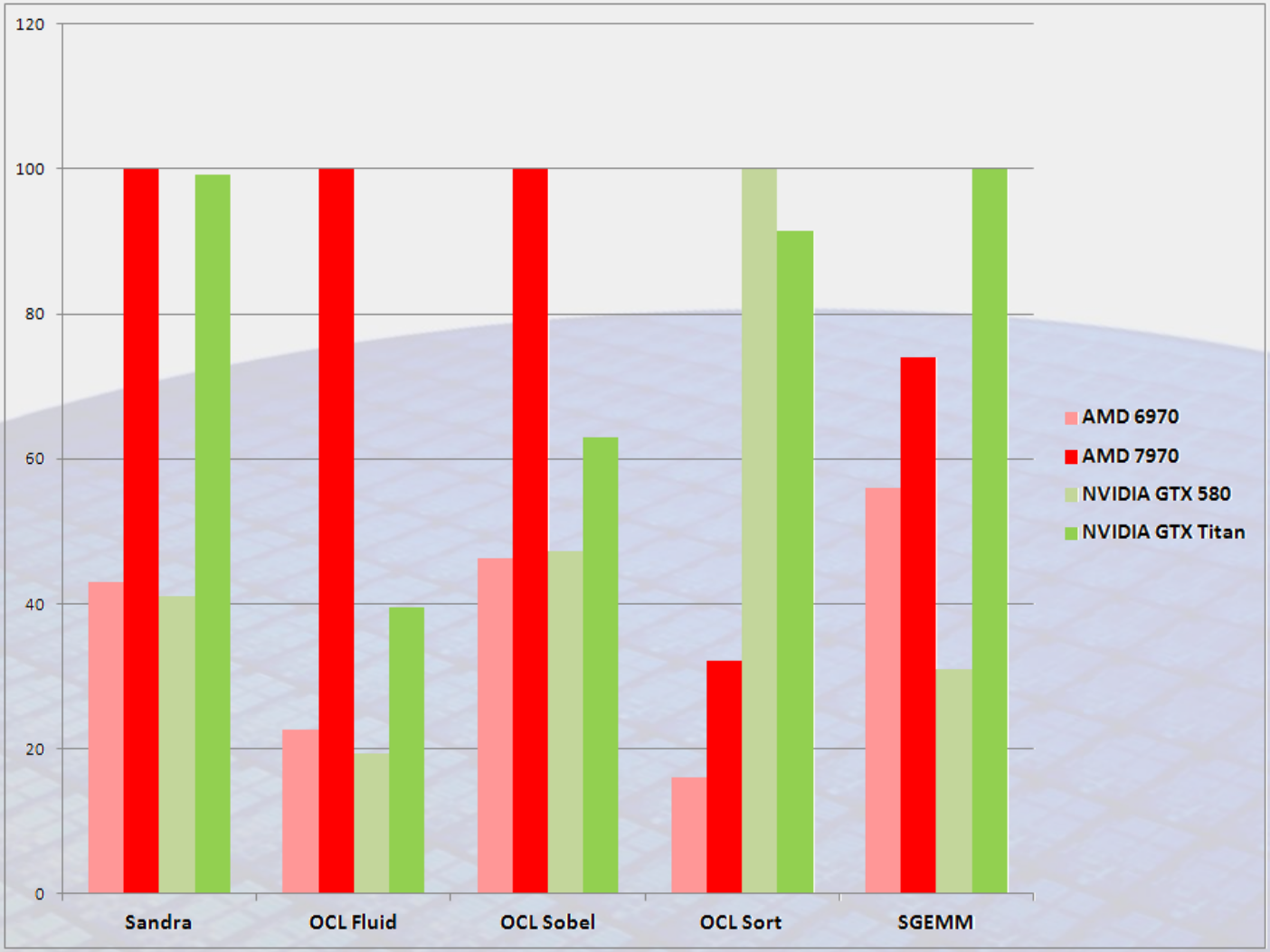



## AMD GCN SIMD architektúra



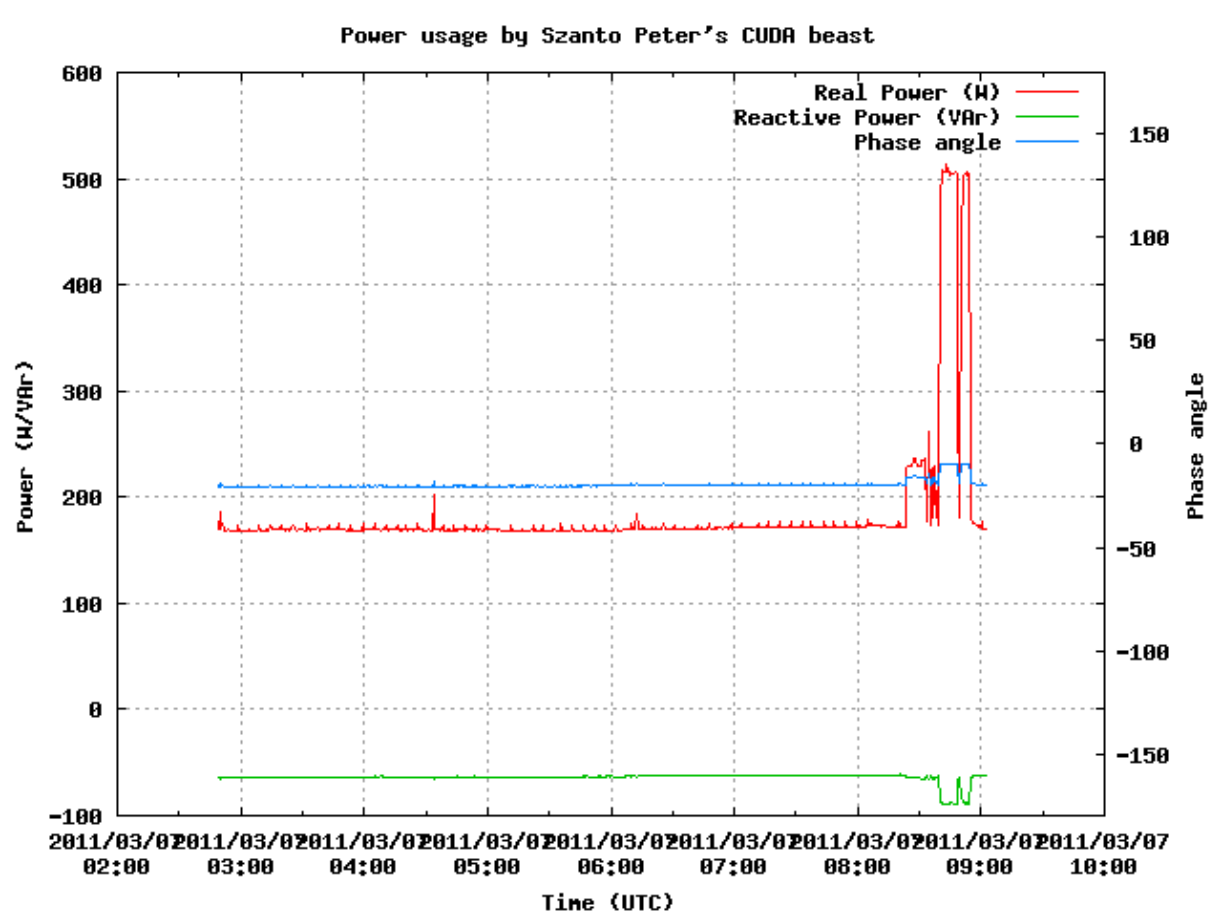
# GPU-k relatív teljesítménye

- Nincs kiforrott, elfogadott benchmark; driver problémák

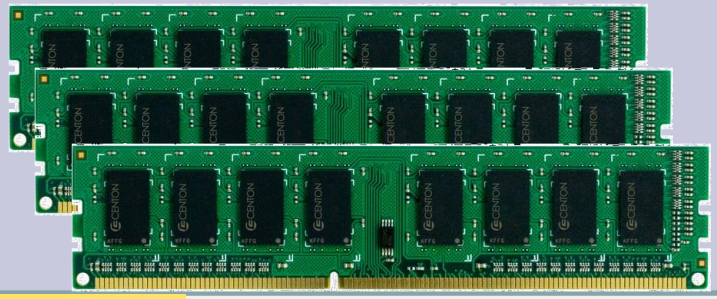


# Fogyasztás…

- Intel Core2 Duo @ 3GHz
- NVIDIA Geforce GTX 480 @800/1600 MHz



# Feldolgozási modell








Bemeneti adatok áttöltése

Eredmények áttöltése

Vezérlés

PCI EXPRESS






Párhuzamos feldolgozás



# CUDA: verziók

- **CUDA Toolkit verzió**
  - SW verziószám, semmi köze a HW-hez
- **Compute Capability: 1.0, 1.1, 1.2, 1.3, 2.0, 2.1**
  - HW képességek verziószáma
  - 1.x: Fermi előtti architektúrák
  - 2.0: GF100 Fermi
  - 2.1: GF100 leszármazottak (GF104, GF108)
  - 3.0: GK104 és társai
  - 3.5: GK110
  - 5.0: Maxwell

# NVIDIA Fermi kártyák

- „Játékos” kártyák – GeForce

| | ALU # | ALU clk | GFLOPS | Mem. BW | Ár |
|---|---|---|---|---|---|
| **GTX 580** | 512 | 1544 | 1581 | 192,4 | 130000 |
| **GTX 570** | 480 | 1464 | 1405 | 152 | 100000 |
| **GTX 480** | 480 | 1400 | 1344 | 177,4 | 70000 |
| **GTX 560 Ti** | 384 | 1645 | 1263 | 128 | 60000 |
| **GTX 460** | 336 | 1350 | 907 | 115,2 | 40000 |
| **GTS 450** | 192 | 1566 | 601 | 57,7 | 25000 |
| **GT 430** | 96 | 1400 | 269 | 28,8 | 15000 |

- **Professzionális grafikus piac: Quadro**
- **GPGPU piac: Tesla – gyors DP csak itt!**
  - Tesla C2050: 448 ALU, 3GB RAM: $2500
  - Tesla C2070: 448 ALU, 6GB RAM: $4000
    - ALU: 1150 MHz; BW: 144 GB/s

# CUDA

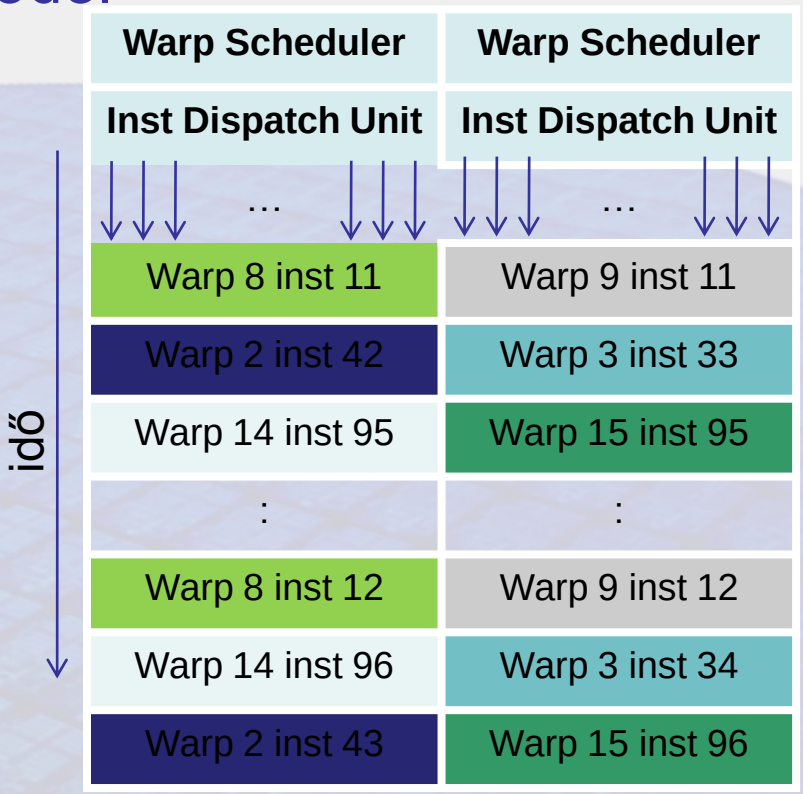
- **C nyelv minimális kiterjesztése**
  - Kernel: a GPU-n futó kód
    - Nincs rekurzió
    - Nincs dinamikus memória foglalás
  - Hoszt: a CPU-n futó kód
    - Driver API: viszonylag alacsony szintű interfész, JIT
    - Runtime API: a Driver API-ra épülő kényelmes programozási felület; (volt emuláció)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

# CUDA szálkezelés

- **„Warp”: 32 szál, ugyanazt az utasítást futtatják időosztásban egy Streaming Multiprocessor-on (SM)**
  - divergens szál: mindkét ág végrehajtása!
- **Thread Block: garantáltan egy SM-en futó szálak tömbje**
  - Adatcsere
  - Szinkronizáció
- **Thread Grid: Thread Block-okból álló tömb**
  - Egymástól gyakorlatilag teljesen független

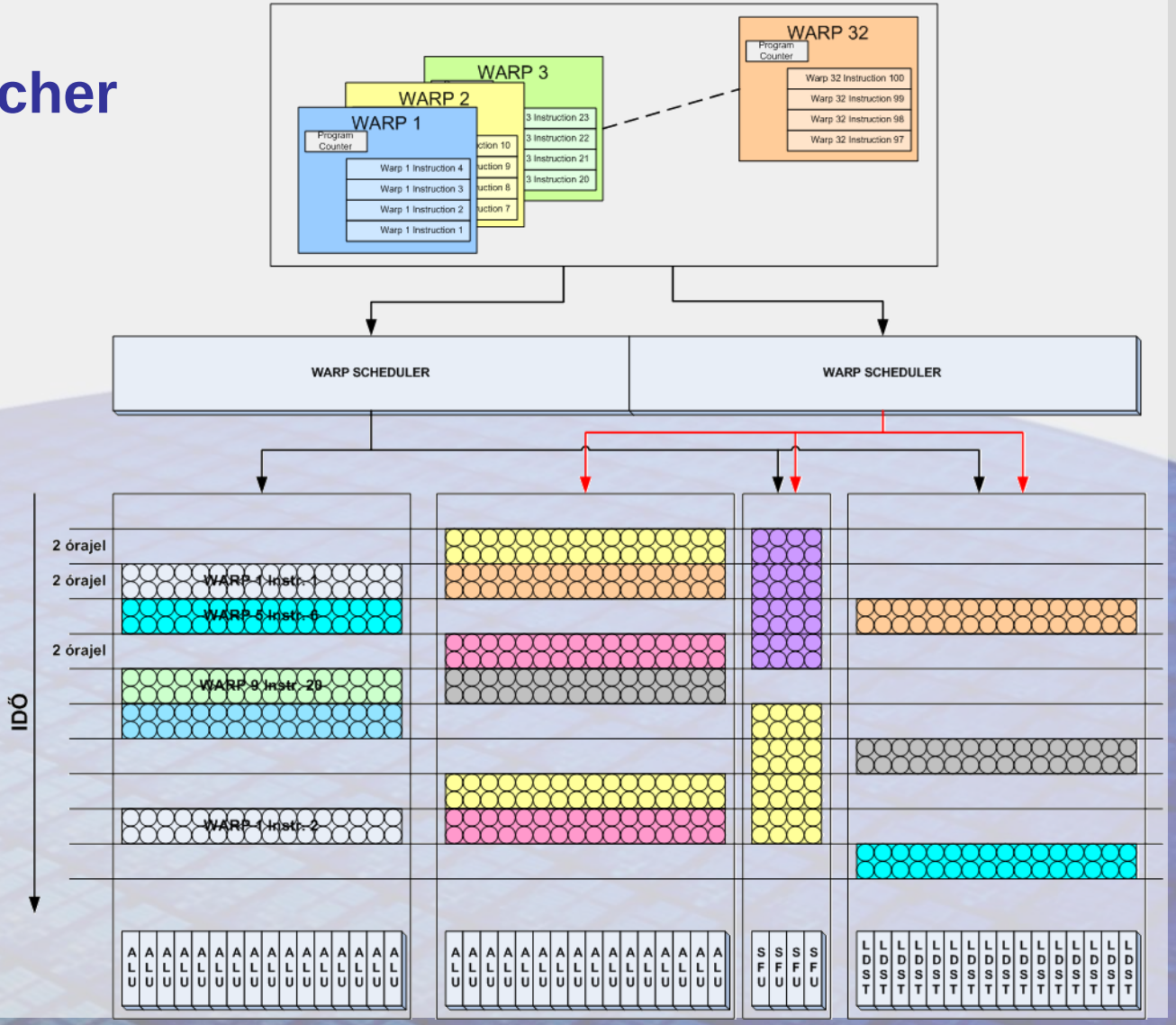
# GF100 – Warp ütemező

- **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**
  - Mindkettő 2-2 utasítást dekódol 1-1 warp-ból
  - 2*2*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 ?)

# 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 → 4*32=128 utasítás ütemezhető
  - A teljes kihasználtsághoz utasítás szintű párhuzamosítás szükséges (mint a GF104 esetében)

# Float teljesítmény

- **GF100**
  - MUL, ADD: 32 flops/SM/clk
  - MADD: 64 flops/SM/clk
- **GF104**
  - ILP=1 MUL, ADD
    - 32 flops/SM/clk
  - ILP=2 MUL, ADD
    - 48 flops/SM/clk
  - MADD
    - Általában 64 flops/SM/clk
    - Kivéve pl. 96 flops:

```
for (1…10000)
  d = d*a;
```

```
for (1…10000)
  d = d + a*b;
```

```
for (1…10000)
  d0 = d0*a0;
  d1 = d1*a1
```

**?: regiszter file sávszélesség limit – 96 ops/MP?**
**?: regiszter allokáció**

```
d[0] = d[0] + a[0]*a[0];
d[1] = d[1] + a[1]*a[1];
```

# INT teljesítmény

- **GF100**
  - MADD, ADD: 32 ops/SM/clk
  - MUL: 16 ops/SM/clk
  - Logikai:  32 ops/SM/clk
  - Egyéb (pl. POPCNT): 16 ops/SM/clk
- **GF104**
  - MADD, ADD: 32 ops/SM/clk
  - MUL: 16 ops/SM/clk
  - Logikai:  32 ops/SM/clk
  - Egyéb (pl. POPCNT): 16 ops/SM/clk

# 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/Kepler/Maxwell esetén cache-elt
  - Van atomi operáció (összes szálra)
- **Local Memory**
  - Thread-ekhez allokált külső memória; Fermi: cache-elt

# SM vs. SMX vs. CU

| | GF100 | GK110 | GM107 | GCN | egység |
|---|---|---|---|---|---|
| ALU | 32 | 192 SP<br>64 DP | 32 | 64 | |
| Max. szál | 1536 | 2048 | 2048 | 2560 | |
| Regiszter | 32k | 64k | 64k | 64k | 32 bit szó |
| Shared Memory | 64<br>16 – 48<br>felosztás | 64<br>16 – 32 –<br>48 felosztás | 64 | 64 | kbyte |
| Cache | | | | 16 | kbyte |
| Read-Only Cache | - | 48 | ? | - | kbyte |
| Max. regiszter / szál | 63 | 255 | 255 | 128 | |
| SP FLOP / clk | 64 | 384 | 64 | 128 | |
| SM BW / clk | 64 | 128<br>(256) | 64 | 128 | byte |
| Cache BW / clk | | | | 64 | byte |
| Regiszter / ALU | 1024 | 341,3 | 2048 | 1024 | szó |
| Sh. Mem / ALU | 1,5 | 0,25 | 2 | 1 | kbyte |
| Cache / ALU | | | | 0,25 | kbyte |
| SM BW / FLOP | 1 | 0,33<br>(0,66) | 1 (?) | 1 | byte/flop |
| Cache BW / FLOP | | | | 0,5 | byte/flop |

# CUDA nyelvi elemek

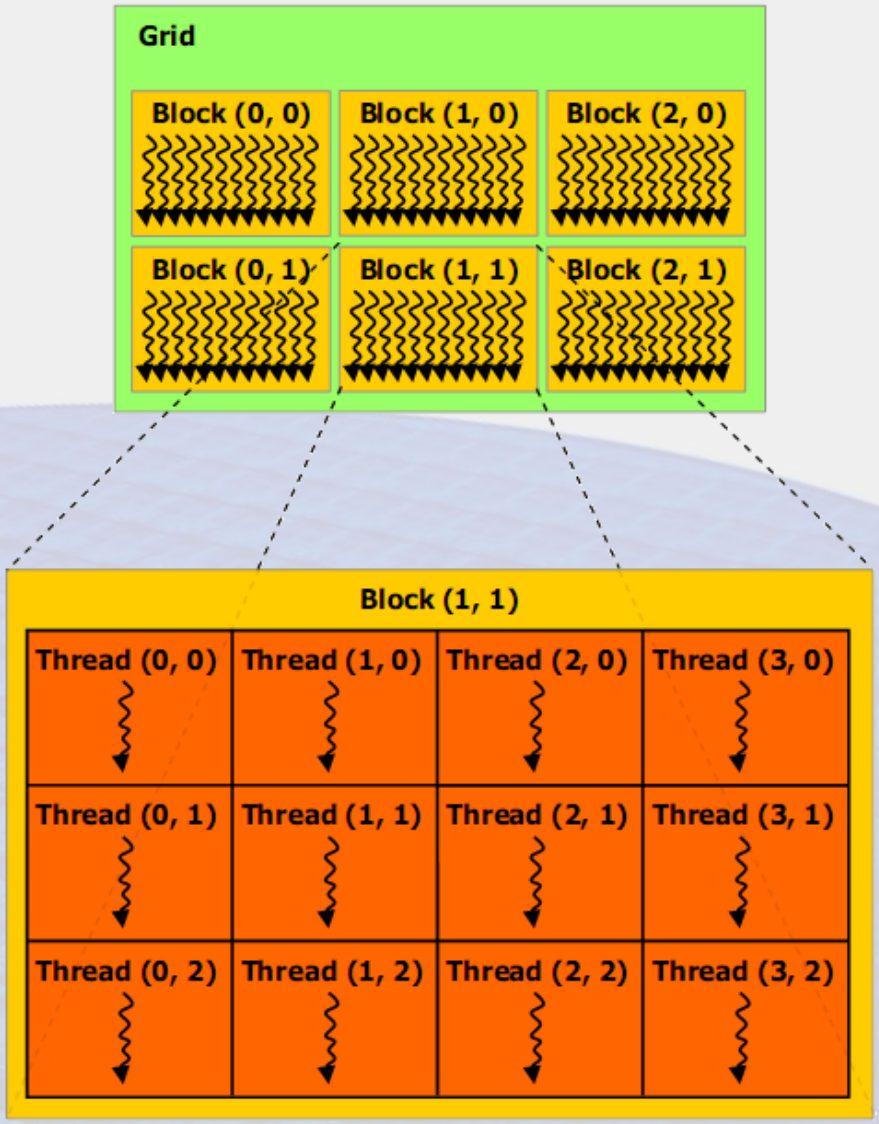
- **__host__ int func(…);**
  - Hoszton futó és onnan hívható függvény
- **__global__ void kernel(...);**
  - hoszt által hívható kernel függvény
- **__device__ void kernel(...);**
  - kernelből hívható kernel függvény
- **__device__ int var;**
  - globális (külső) memóriában deklarált változó
- **__shared__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

# CUDA nyelvi elemek

- **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 – 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 (Fermi: 3D)**
  - legfeljebb 2D: {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

# 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)
      - SM1.3: 16k; SM2: 32k
    - Shared Memory mérete
      - SM1.3: 16kB; SM2: 16kB vagy 48kB

# Thread Block – SM

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

# 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

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% | |

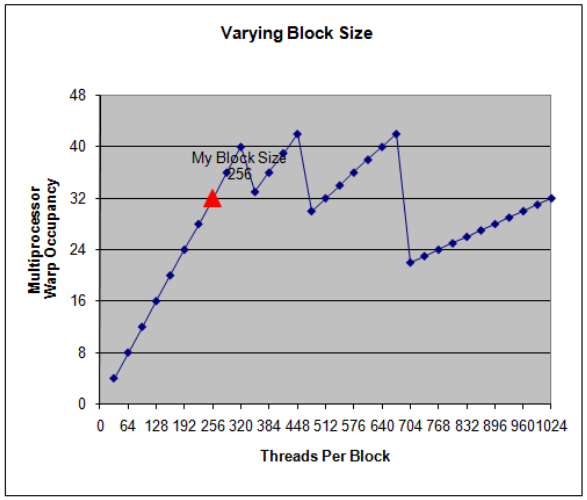
| Physical Limits for GPU Compute Capability: | 2,1 |
|---|---|
| Threads per Warp | 32 |
| Warps per Multiprocessor | 48 |
| Threads per Multiprocessor | 1536 |
| Thread Blocks per Multiprocessor | 8 |
| Total # of 32-bit registers per Multiprocessor | 32768 |
| Register allocation unit size | 64 |
| Register allocation granularity | warp |
| Shared Memory per Multiprocessor (bytes) | 49152 |
| Shared Memory Allocation unit size | 128 |
| Warp allocation granularity (for register allocation) | 2 |
| Maximum Thread Block Size | 1024 |

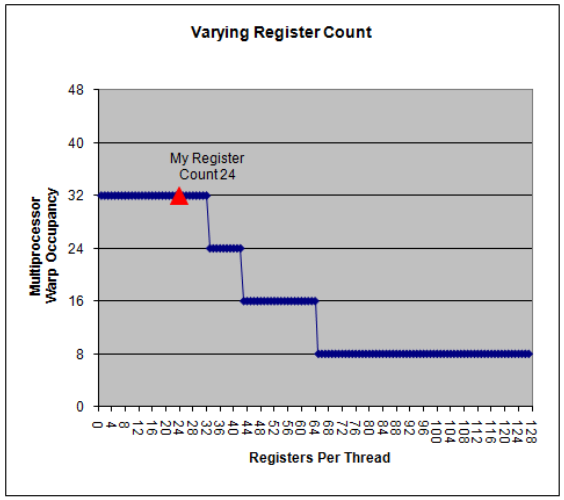
| Allocation Per Thread Block | |
|---|---|
| Warps | 8 |
| Registers | 6144 |
| Shared Memory | 12032 |

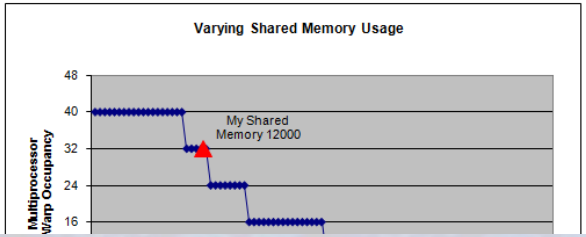
These data are used in computing the occupancy data in blue

| 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.



# Kernel futtatás

- dim3: beépített változó típus, legfeljebb 3 eleme van (x,y,z) – a nem használt 1
- 16x16 szálas Thread Block 8x8-as grid-ben → összesen (16*16)*(8*8)=16384 szál

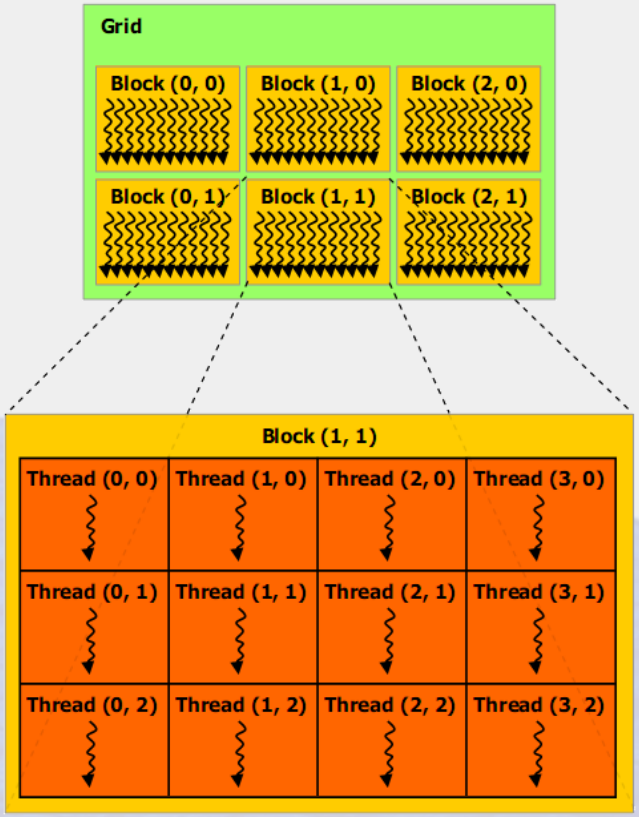
```
dim3 thrBlock(16, 16);
dim3 thrGrid(8, 8);

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 szálat
- **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 – kernel

- Kernel visszatérési értéke: mindig void
- A, B, C: a videókártya memóriájában foglalt memóriaterület pointere
- N: a futtatott szálak száma

```
__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 – 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 *gA, *gB, *gC;
   cudaMalloc((void**)&gA, N*sizeof(float));
   cudaMalloc((void**)&gB, N*sizeof(float));
   cudaMalloc((void**)&gC, N*sizeof(float));
……..……..……..……..……..……..
}
```

**Hoszt memória foglalása**

**GPU memória foglalása**

# Hoszt memória foglalás

- **Sztenderd C függvények: malloc(), _aligned_malloc()**
- **cudaError_t cudaMallocHost(void **ptr, size_t size)**
  - Nem lapozható memória
- **cudaError_t cudaHostAlloc(void **ptr, size_t size, unsigned int flags)**
  - cudaHostAllocDefault
    - nem lapozható memória
  - cudaHostAllocPortable
  - cudaHostAllocMapped
    - GPU részéről is elérhető: cudaHostGetDevicePointer()
  - cudaHostAllocWriteCombined
    - gyors PCIe írás, lassú CPU olvasás
- **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, megfelelő határra helyezve
- **cudaError_t cudaMalloc3D( )**
  - 3D tömb megfelelő határra helyezve

# Memória másolás

```
.....

   int size = N*sizeof(float);
   cudaMemcpy(gA, hA, size, cudaMemcpyHostToDevice);
   cudaMemcpy(gB, hB, size, cudaMemcpyHostToDevice);

   int thrBlock = 256;
   int thrGrid = N/thrBlock;
   VecAdd<<<thrGrid, thrBlock>>>(gA, gB, gC);

   cudaMemcpy(hC, gC, size, cudaMemcpyDeviceToHost);
   cudaFree(gA); cudaFree(gB);
   cudaFree(gC);
}
```

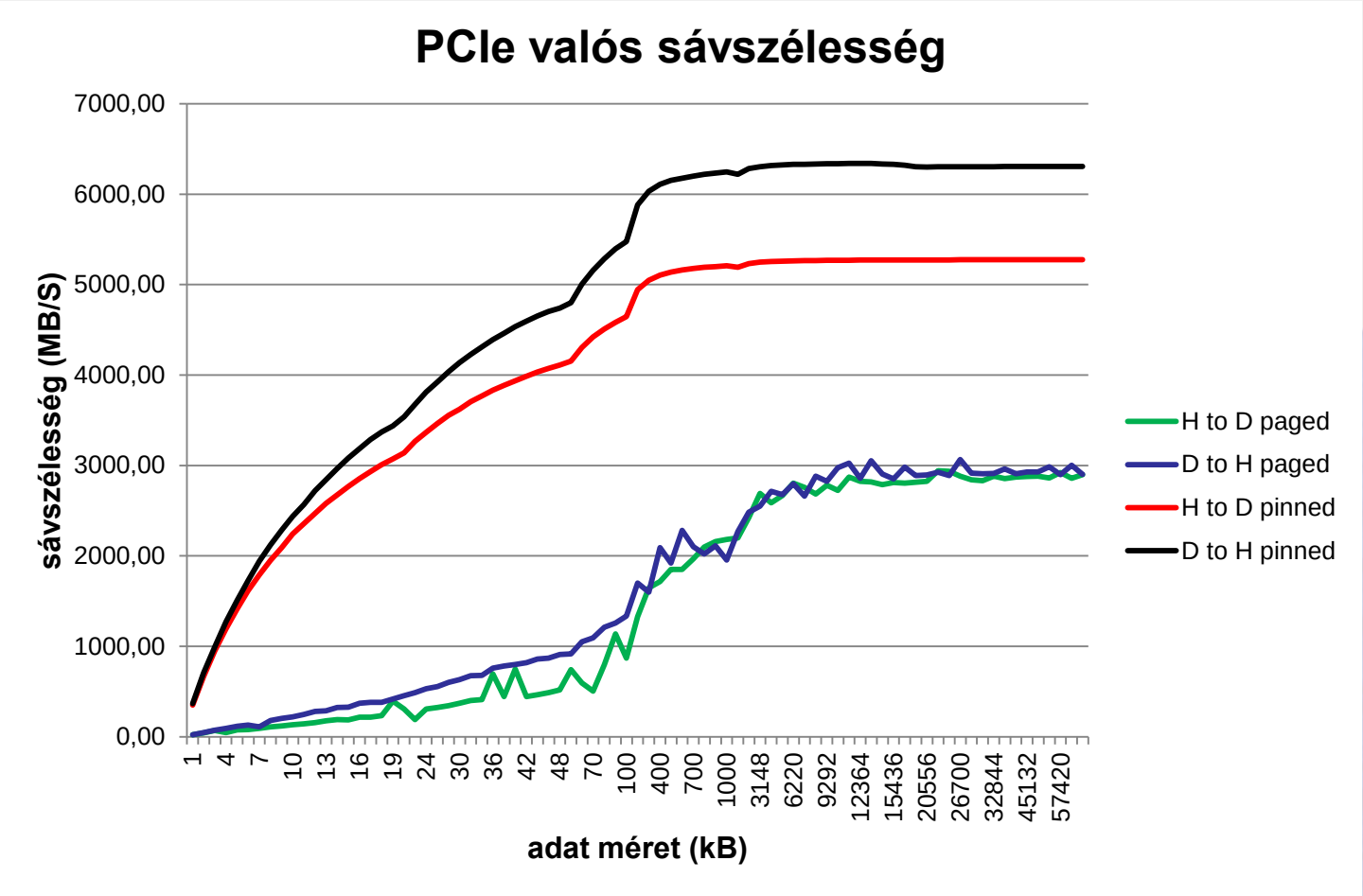
**Bemeneti adatok másolása**

**Kernel futtatása**

**Eredmények másolása**

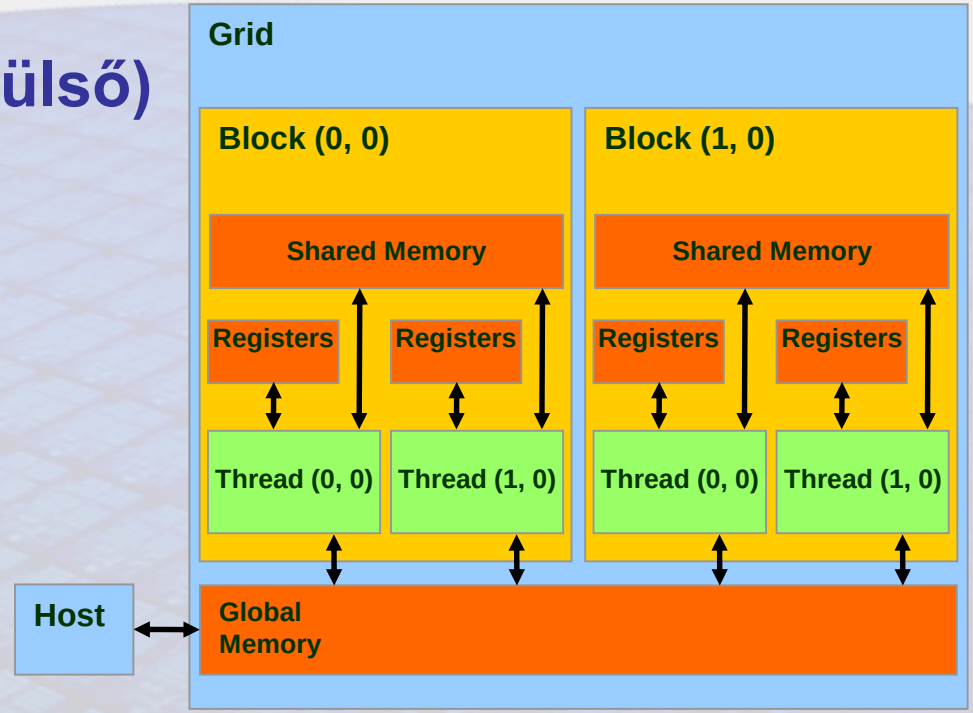
# PCIe átviteli sebesség

- Elsősorban a chipset-től függ!



# GPU memóriák

- **Szálanként**
  - Regiszter
  - Local Memory (helyileg: külső memóriában!)
- **Thread Block-onként**
  - Shared Memory
- **Közös: Global Memory (külső)**
  - Local Memory
  - Constant Memory
  - Texture Memory

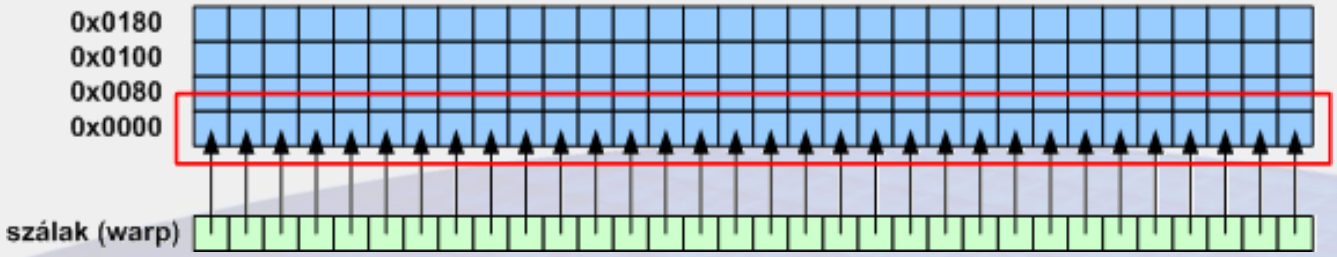


# GPU globális memória

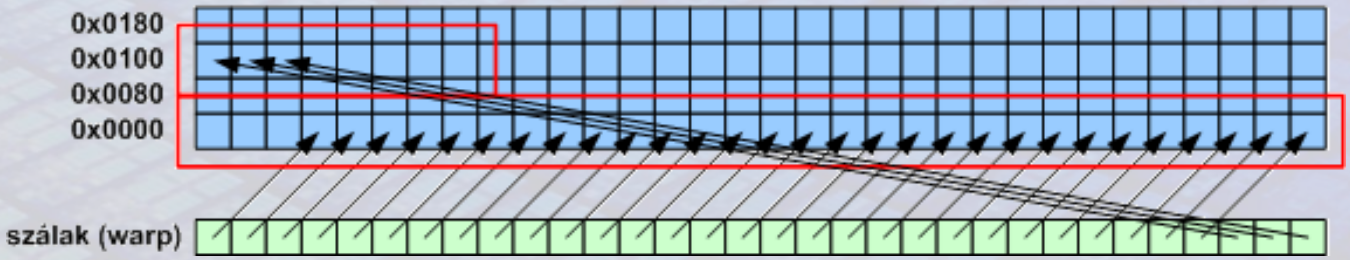
- **Szegmens méret (és határ)**
  - 32 byte 1 byte-os hozzáférésnél
  - 64 byte 2 byte-so hozzáférésnél
  - 128 byte 4 byte-os hozzáférésnél
- **Minimum átviteli méret: 32 byte**
- **Nem használt adatok olvasása!**
- **Memória műveletek száma fél-warp-onként dől el**
  - Fél-warp által használt memória szegmensek száma
- **Fermi**
  - Global Memory és Local Memory hozzáférések cache-elve (mindkét irányban) → hatalmas előny

# Memória hozzáférési minták (1)

- **Egyszerű minta: szekvenciális hozzáférés**
  - Báziscím 128 byte többszöröse
  - 1 db 128 byte-os RD: 70 GB/s másolás (GF104)

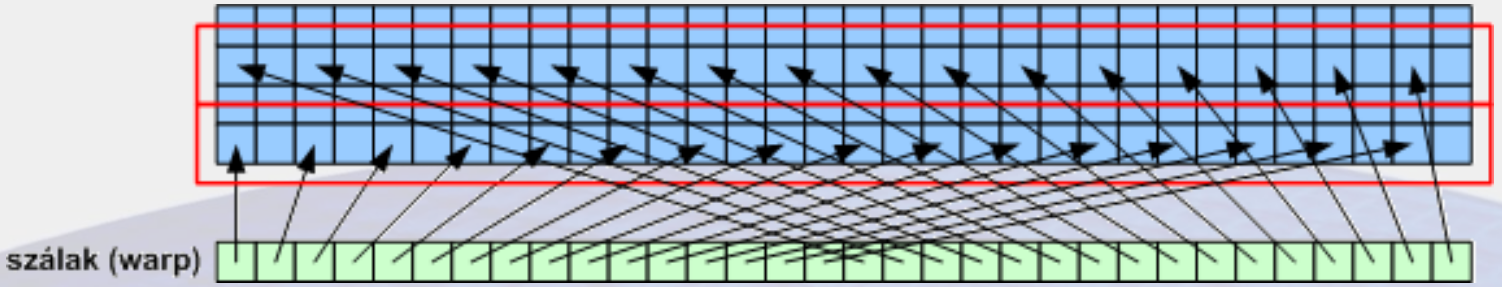



- **Eltolt hozzáférés**
  - 1x128 byte + 1x32 byte
  - GF104: 70 GB/s; CC1.x: eltolás mértékétől függő lassulás



# Memória hozzáférési minták (2)

- **„Stride” hozzáférés**
  - Minden szál a sorszáma*S címet éri el
  - Pl. S=2




  - 2x128 byte hozzáférés → a beolvasott adatok fele viszont NEM kerül felhasználásra → effektív sávszélesség S-ed része
  - Nem mindig elkerülhető – pl. 2D tömb egy oszlopának elemein lépkedünk végig → máshogy kell megkerülni

# Lokális & konstans memória

- **Lokális memória: fizikailag külső memóriában**
  - Automatikusan minimális számú memória művelettel jár
  - Fermi esetén cache-elt
  - Regiszterek átmeneti tárolás → sokat lassíthat!
- **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

- **32 bank a warp 32 szálához (CC1.x: 16 – fél-warp)**
  - 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!**

# Textúrák

- **Textúra memória csak olvasható, cache-elt**
  - Írásnál nincs cache koherencia!
  - Fermi esetén írható, de nem koherens
  - 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;
```

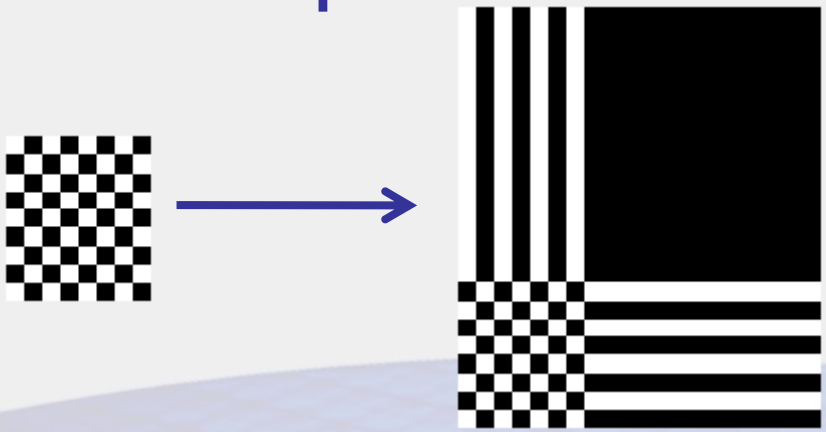
- **TYPE: a texel típusa**
  - INT, FLOAT (ill. 1/2/4 komponensű vektor típusok)
- **DIM: a textúra dimenziója**
  - 1, 2 vagy 3
- **ReadMode: az olvasás során visszaadott adattípus**
  - cudaReadModeElementType: az olvasott adat a texel típusának megfelelő
  - cudaReadModeNormalizedFloat: automatikus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

# Textúra referencia paraméterek

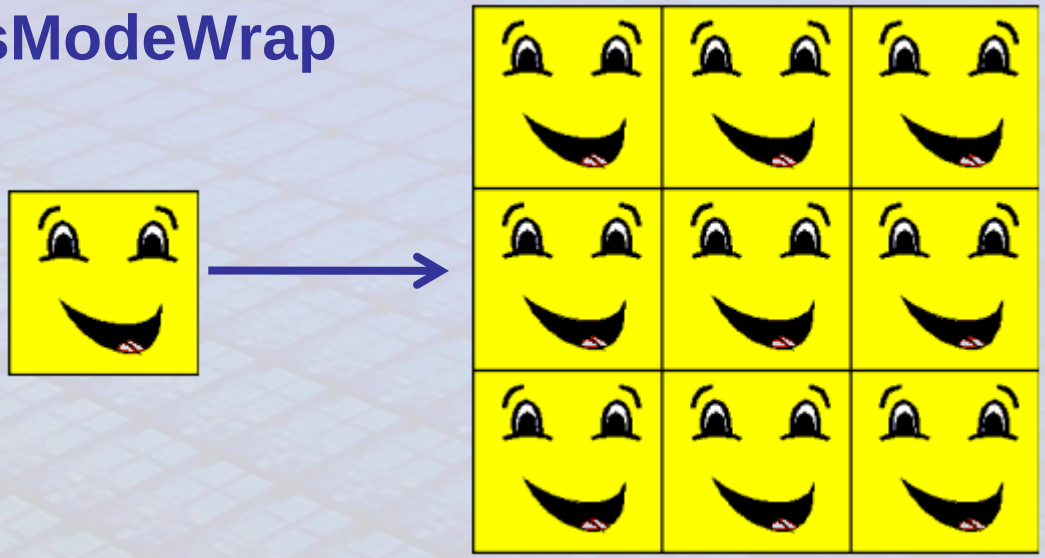
- **Címzési mód**
  - Normalizált: a textúra texelei a [0, 1] tartományba eső értékekkel címezhetők
  - Nem-normalizált: a texelek a textúra szélességnek és magasságnak megfelelő egész értékekkel címezhetők
- **filterMode**
  - cudaFilterModePoint: nem texel középpontra eső cím esetén a legközelebbi texelt adja vissza
  - cudaFilterModeLinear: lineáris interpoláció a legközelebbi 2/4/8 texel használatával (csak FP ReadMode esetén)
- **channelDesc: textúra olvasási típus megadás**
  - cudaChannelFormatKindSigned: előjeles egész
  - cudaChannelFormatKindUnsigned: pozitív egész
  - cudaChannelFormatKindFloat: lebegőpontos

# Textúra címzési módok

- cudaAddressModeClamp



- cudaAddressModeWrap



# Textúra példa – hoszt (1.)

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

   cudaArray* cuArray;

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

Referencia

Komponens leíró

GPU memória

Textúra memória írása

# Textúra példa – hoszt (2.)

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

    cudaBindTextureToArray(tex0, cuArray,
         channelDesc);


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

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

**Paraméter beállítás**

**Referencia – memória összerendelés**

**Textúra memória írása**

**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;
}
```

# Optimalizálás

- **Hoszt – GPU átvitel**
  - 3 – 5 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: Fermi kivételével 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

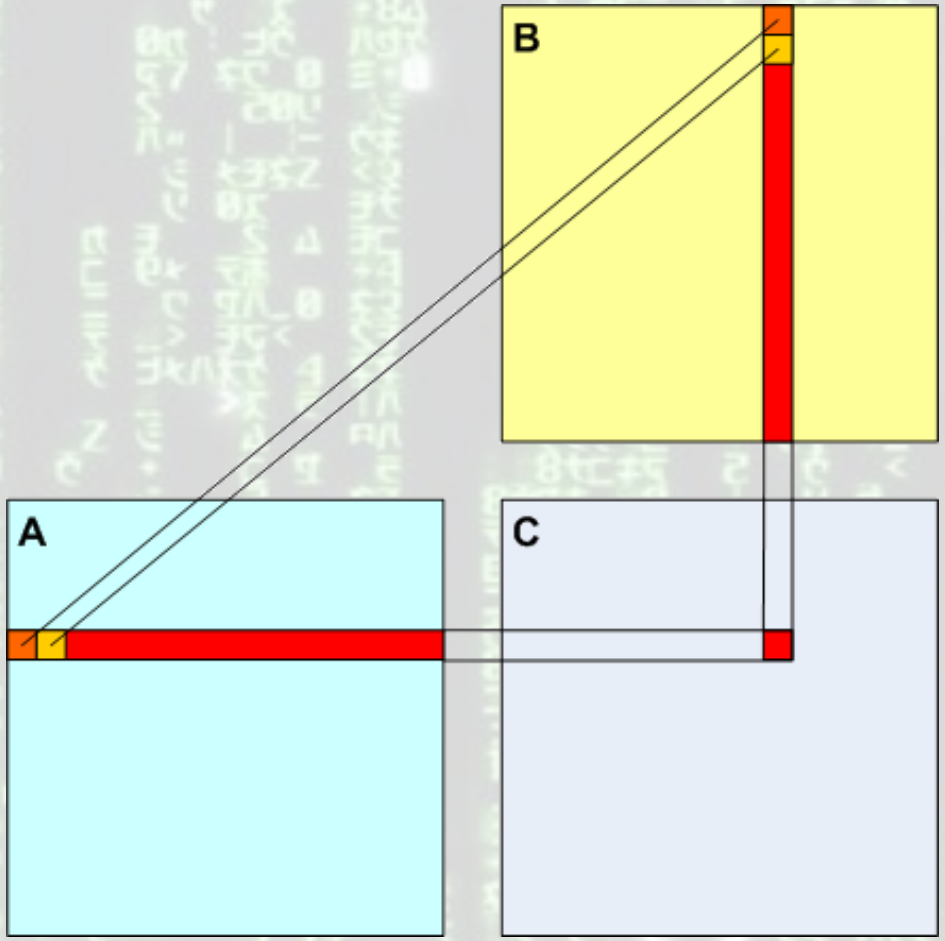
# Műveleti sebesség

- **Művelet/órajel/multiprocesszor (1.x: 8 ALU, 2.0: 32 ALU)**

| | 1.x | 2.0 |
|---|---|---|
| SP +, *, madd | 8 | 32 |
| DP +, *, madd | 1 | 16 |
| INT (+, logikai, shift, cmpr) | 8 | 32 |
| 24-bit * | 8 | Több utasítás |
| 32-bit * | Több utasítás | 32 |
| __sin, __log, __exp | 2 | 4 |

# Mátrix szorzás

- **Két, NxN-es mátrix szorzata**
  - N*N*N MADD művelet
  - N*N*N*2 load
  - N*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



# MMUL: C kód

- Triviális C kód

~9% hatékonyság

```
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;
  }
}
```

# MMUL: GPU (1)

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

GF104: 75 Gfl./s, 8%
GF100: 120 Gfl./s, 8%

```
__global__ void kernel_gl(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;

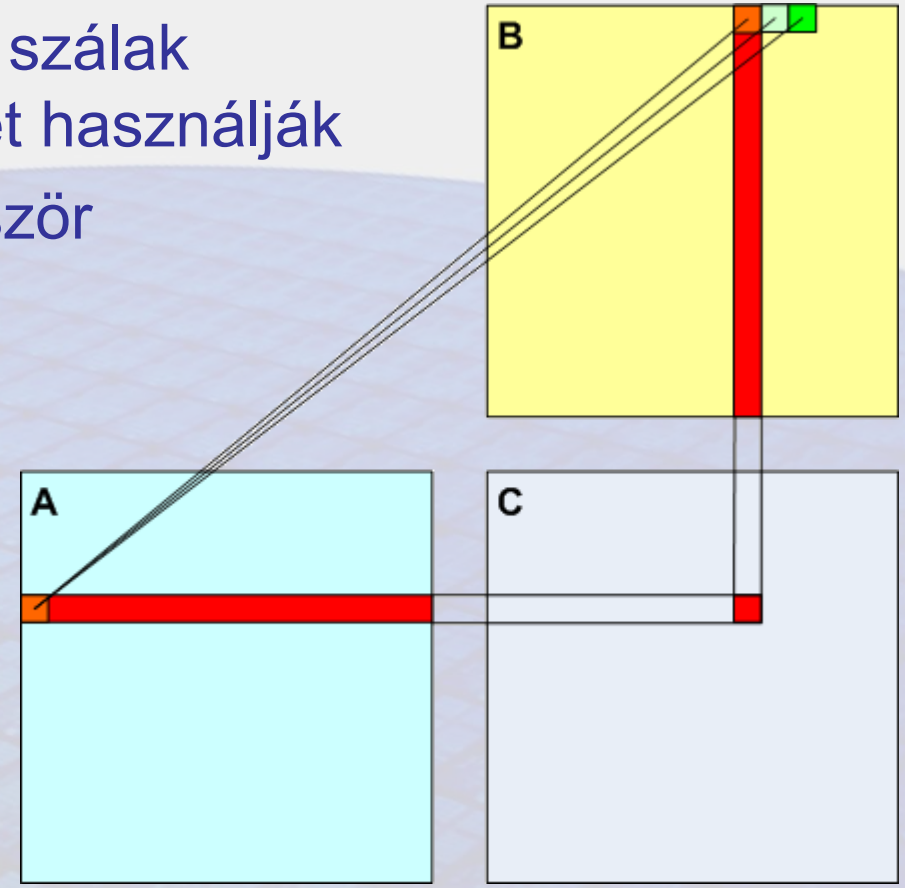
    #pragma unroll 32
    for (int i = 0; i<N; i++)
    {
        sum += A[row*N+i] * B[i*N+col];
    }
    C[row*N+col] = float(sum);
}
```

# 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



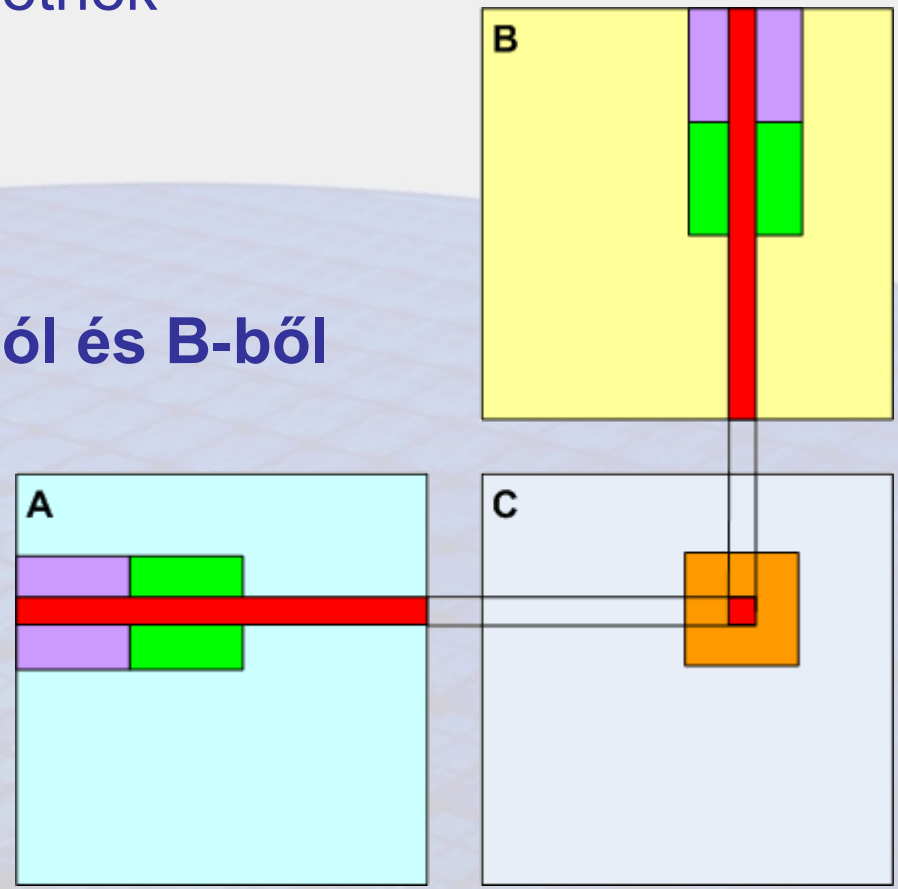
- **Shared Memory**



# MMUL: GPU (3)

- **16x16 Thread Block C 16x16 részével foglalkozik**
  - B-ből 16 széles részt kell beolvasni
  - A-ból a memória burst méretnek megfelelőt → 16

- **16x16 blokkot olvasunk A-ból és B-ből**
  - 16*16*4*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];

  __syncthreads();

.....
```

16x16 blokkonként lépkedünk végig

Shared Memory deklaráció

Minden szál A és B egy-egy elemét tölti be

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

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;
   }


   __syncthreads();
}

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

**GF104: 125 Gfl./s, 14%**
**GF100: 250 Gfl./s, 17%**
**GK107: 71 Gfl./s, 8%**

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

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

# 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
    - ALU órajel felével jár

    → **16 olvasás / ALU clk**
  - 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 Gfl./s, 20%**
**GF100: 392 Gfl./s, 24%**
**GK107: 102 Gfl./s, 12%**

# MMUL: 4-8 kimenet / szál

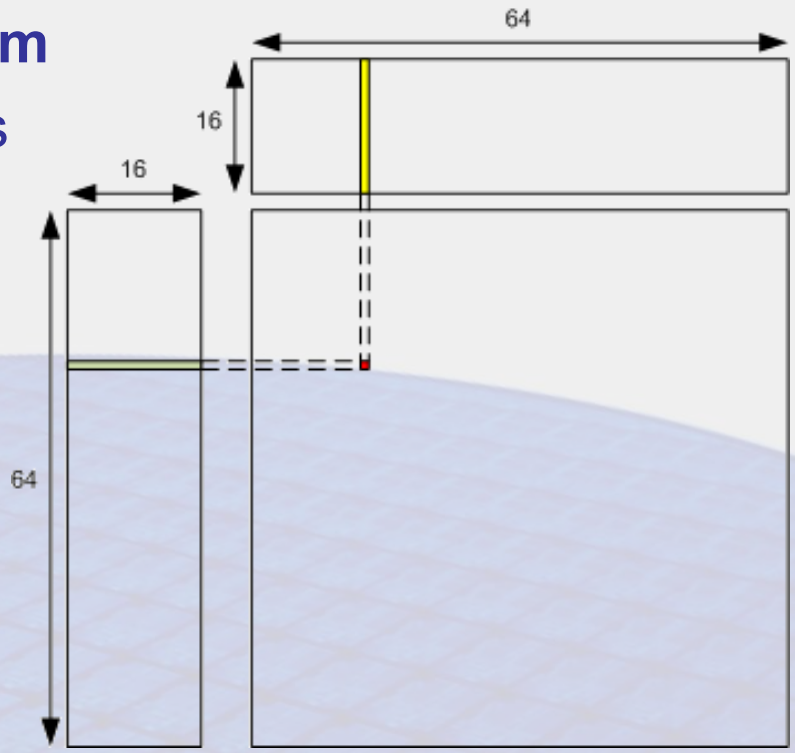
- **Továbbra is 32x32 kimeneti blokk**
  - 4 kimenet: 32x8 Thread Block
  - 8 kimenet: 32x4 Thread Block
- **Kihasználtság (occupancy)**
  - 32 regiszter/szál → 67% occup.
  - 47 regiszter/szál → 42% occup.
- **További kimenet/szál növelés ezen a módon nem jó**
  - Thread Block mérete nagyon lecsökken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

**GF104: 225 Gfl./s, 25%**
**GF100: 492 Gfl./s, 34%**
**GK107: 132 Gfl/s, 16%**

**GF104: 260 Gfl./s, 29%**
**GF100: 553 Gfl./s, 39%**
**GK107: 154 Gfl./s, 18%**

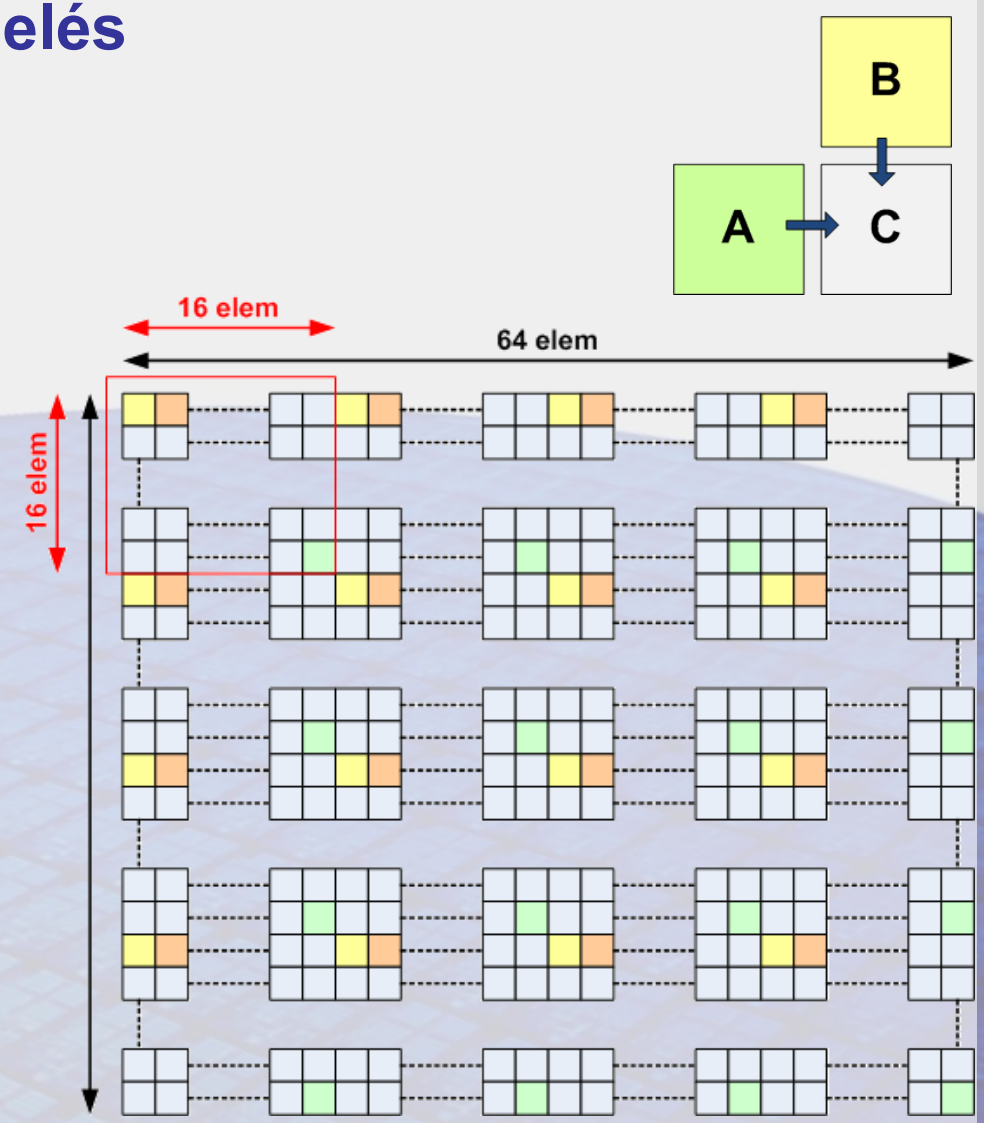
# MMUL v3 (1)

- **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: 16x64=4 kbyte
  - B SMEM: 64x16=4 kbyte
- **Thread block: 16x16 → minden szál 4x4 kimenetet számít**
  - Mind x, mind y irányban 16-os lépésközzel → azaz az egymás melletti 16 érték egymást követő 16 szálhoz tartozik (lsd. következő ábra)



# MMUL v3 (2)

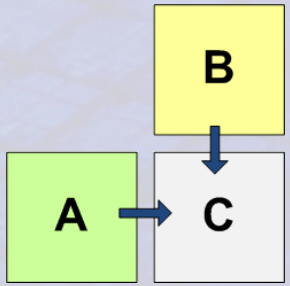
- **Kimenet – szál összerendelés**
  - Az azonos színezett elemeket azonos szál számítja
- **A mtx 1x64 rész-oszlopa és B mtx 64x1 rész-sora alapján minden elem 1-1 rész-szorzata számítható**
- **SMEM 16 oszlopot (A) és 16 sort (B) tartalmaz**



# MMUL v3 (3)

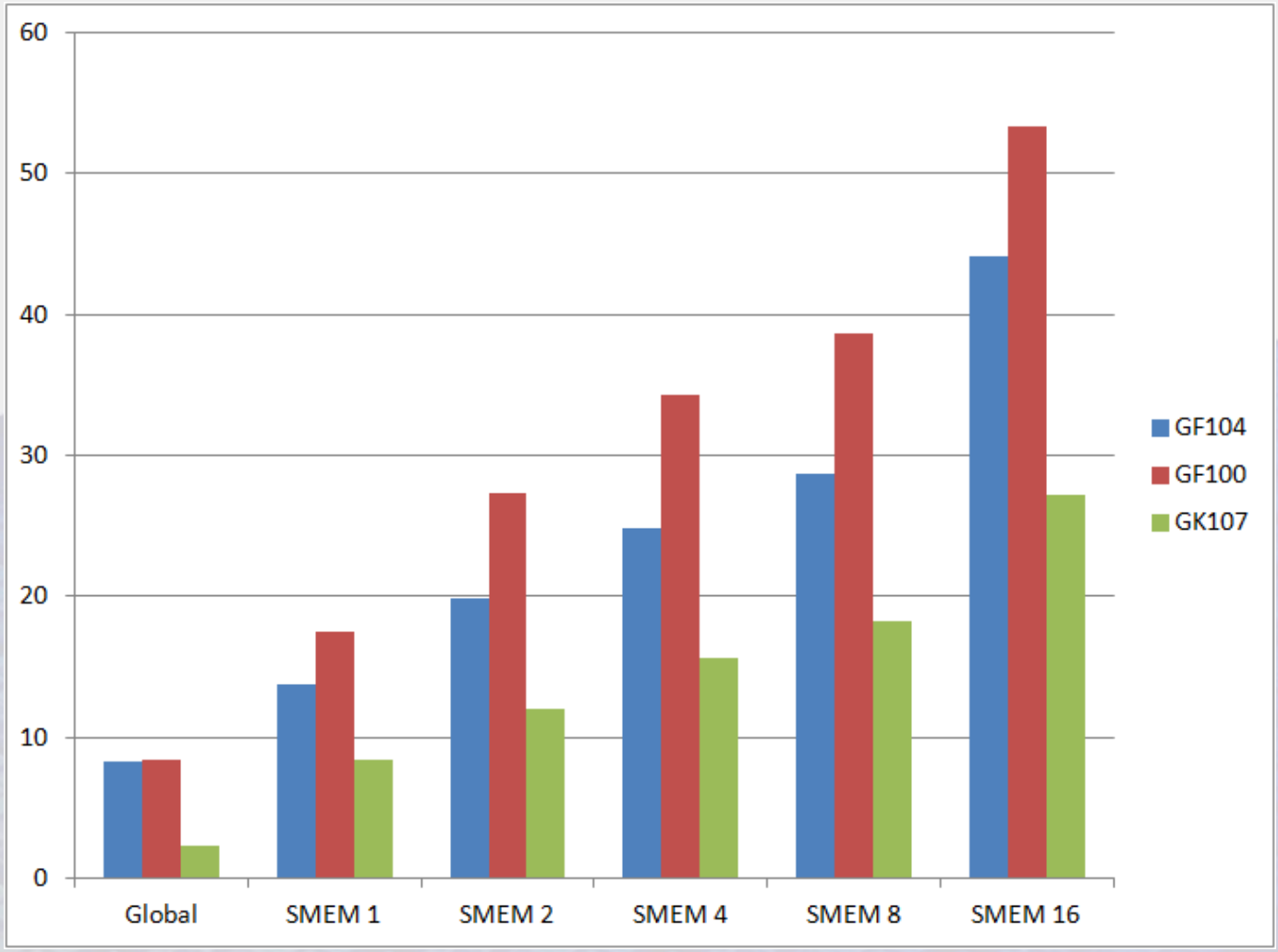
**GF104: 400 Gfl./s, 40%**
**GF100: 765 Gfl./s, 53%**
**GK107: 230 Gfl./s, 27%**

- **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 töltenie!
- **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



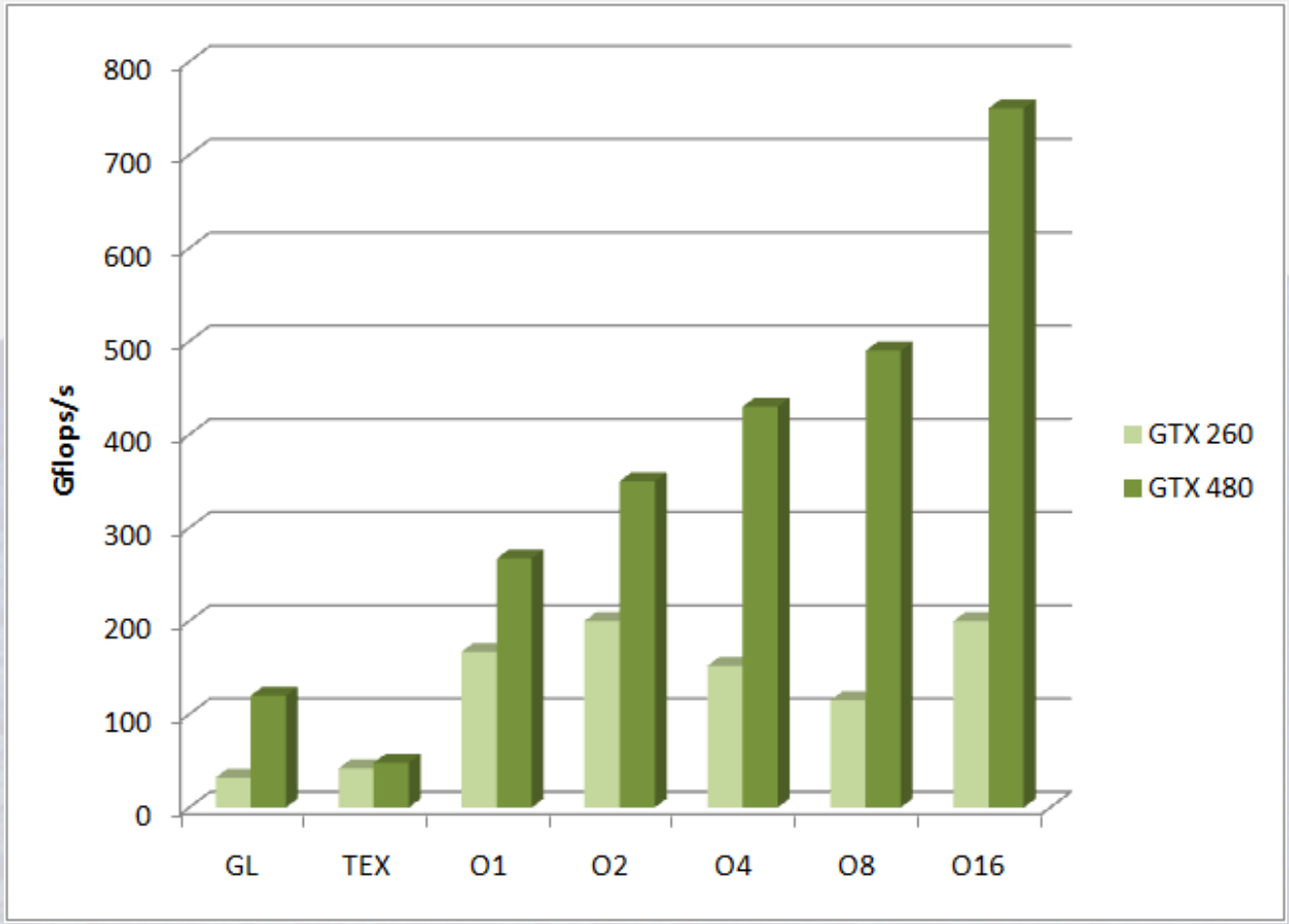
# Mmul teljesítmény (%)

- Elvi számítási teljesítmény %-ában



# GTX 260 vs. GTX 480

- GeForce GTX 260: CC 1.3, 216 ALU, 1.2 GHz, 64 TEX
- GeForce GTX 480: CC 2.0, 480 ALU, 1.6 GHz, 60 TEX



# FIR szűrő (1. gyakorlat)

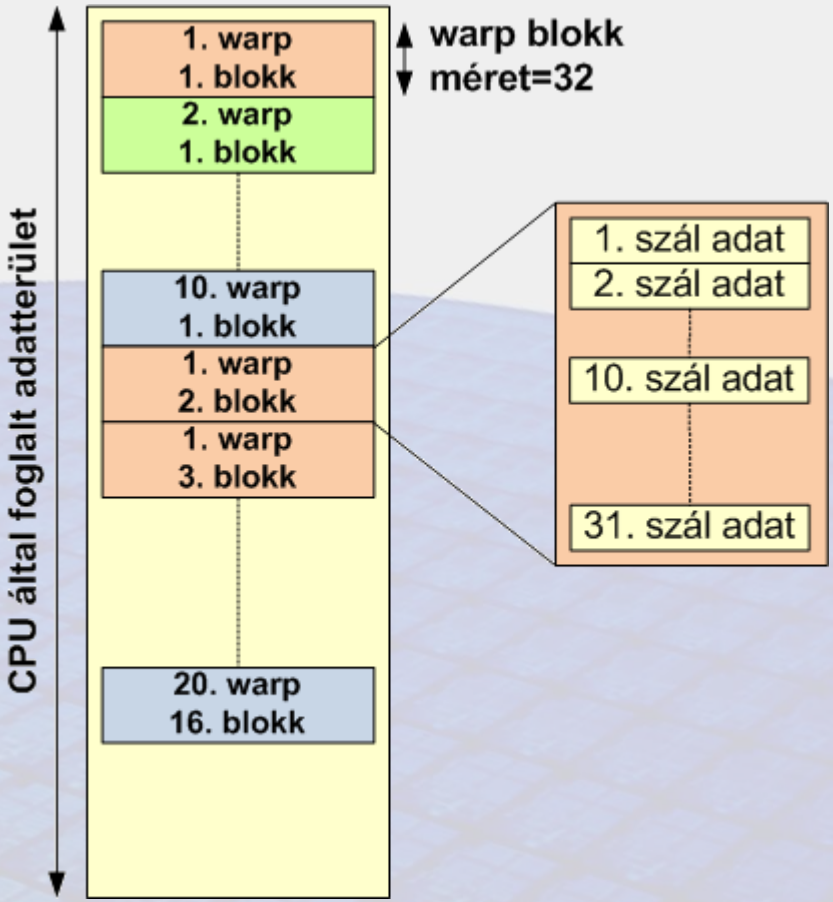
| Futási idő (sec) | Core i5 @2,5 GHz | |
|---|---|---|
| CPU (C kód) | 3,6 | |
| CPU (C + OpenMP) | 1 | |
| CPU (SSE + OpenMP) | 0,28 | |
| Kernel futási idő (sec) | GTX 480 (448 ALU) | GTX 540M (96 ALU) |
| CUDA kernel globális memória | 0,02 | 0,18 |
| CUDA kernel shared memória (1) | 0,018 | 0,12 |
| CUDA kernel shared memória (2) | 0,012 | 0,09 |
| CUDA kernel shared memória (3) | 0,009 | 0,06 |
| CUDA kernel shared memória (4) | 0,008 | 0,06 |

# Ismeretlen kimeneti adatméret

- **A feladatok jelentős részénél a bemeneti adatmennyiség egyértelműen meghatározza a kimeneti adatmennyiséget (pl. mtx mul, képfeldolgozás, FFT, …) → OK**
- **Egyes esetekben a szálak által generált kimeneti mennyiség nem determinisztikus**
  - CPU egyetlen memória területet foglal (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

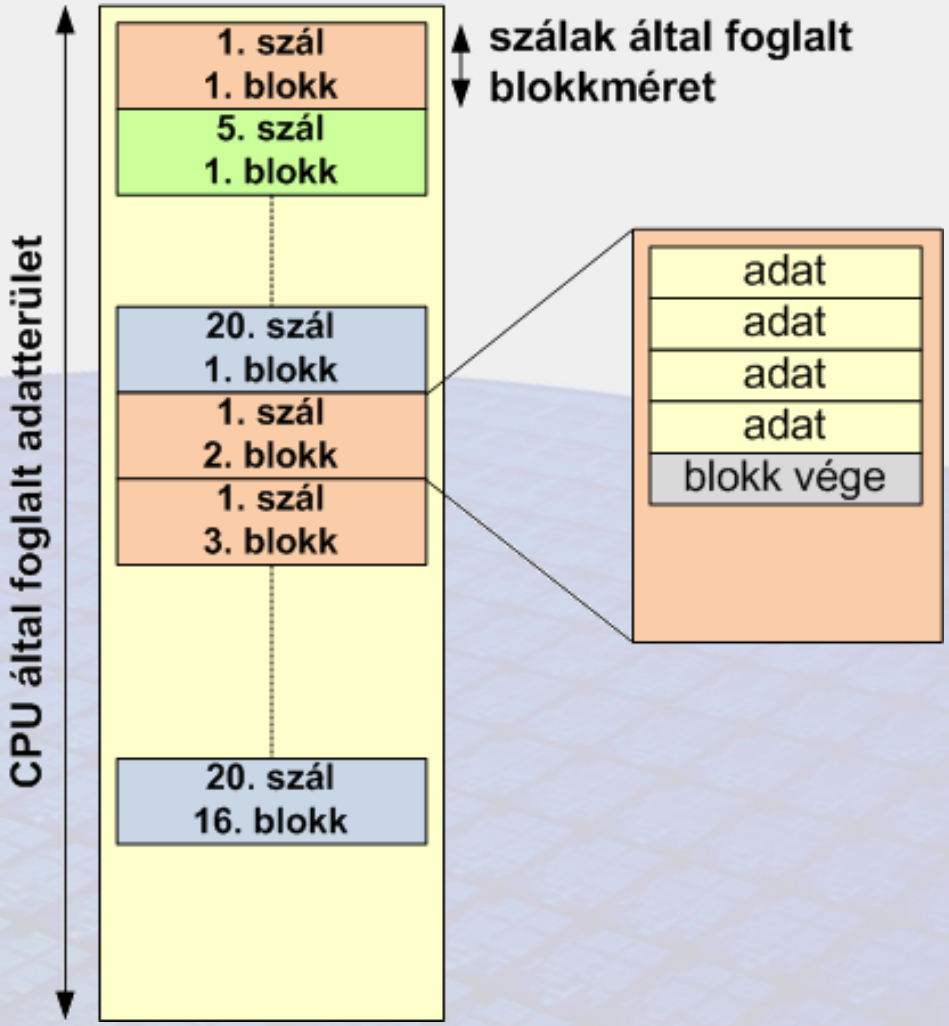


# Dinamikus blokk v1

- **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 foglalás
  - Túl nagy: pazarló
  - ~átlagos kimeneti szám
- **GPU → PC átvitel**
  - Csak a használt méret

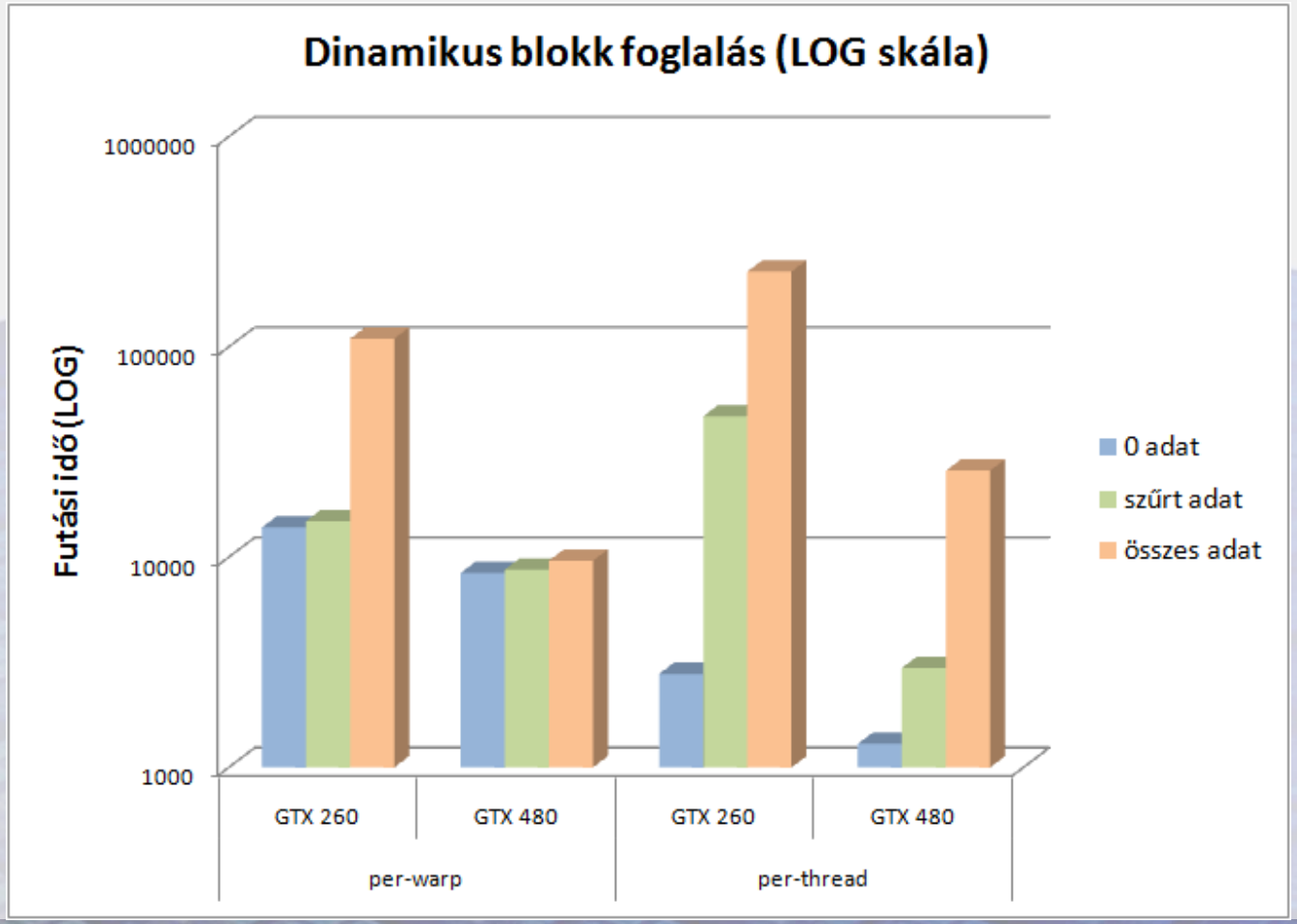


# Blokk foglalás – global atomic

- **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!
    - Túl kicsi blokkméret esetén túl gyakori
- **Hátrány: blokkon belül az egymást követő címeket ugyanaz a szál írja → nem coalesced**
  - Fermi cache itt rengeteget segít

# Sebesség

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



# Stream

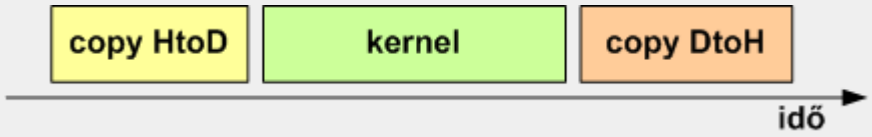
- **Stream: parancsok sorozata**
  - Alapértelmezett stream: 0
- **Konkurrens hoszt – GPU végrehajtás**
  - CudaMemcpyAsync();

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

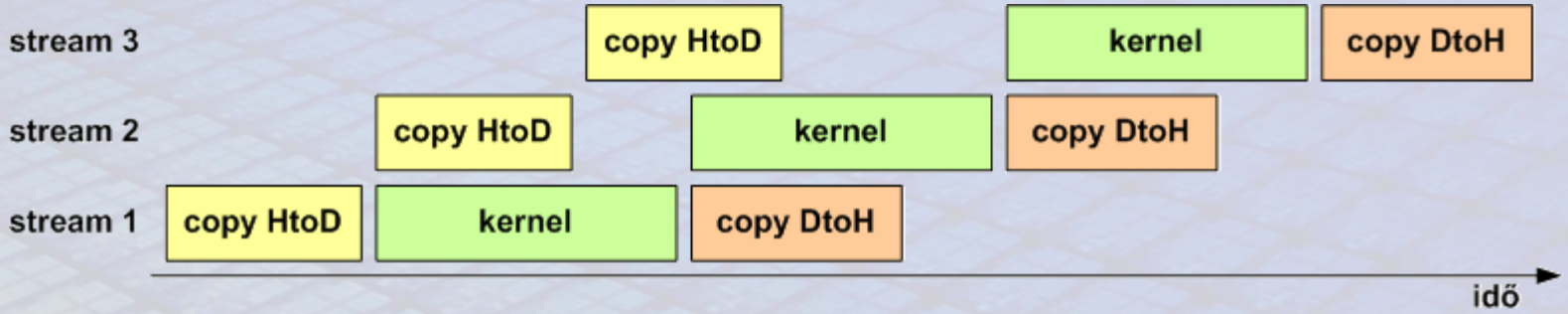
- **Egyes GPU-kkal lehetséges a memória műveletek és a kernel futtatás párhuzamosítása**
  - Nem 0. stream
  - Nem lapozható memória
  - deviceOverlap státusz

# 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: C = 16 ms
  - Kernel futási idő: K = 32 ms
  - Teljes futási idő: C + K = 16 + 32 = 48 ms
- **4 stream → stream-enként C/4 és K/4**
  - Futási idő: C/4 + (K/4)*4 = 4 + 32 = 36 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)
);
```