

ELTE | IK

3. Gyakorlat

CUDA Programozás

Warpok, vezérlési divergencia és párhuzamos redukció

Balázs Endre Szigeti

Algoritmusok és Alkalmazásaik Tanszék
Informatikai Kar, Informatikatudományi Intézet
Eötvös Loránd Tudományegyetem

November 6, 2026

Célkitűzés

- Megérteni, hogyan futnak a CUDA szálak **SIMD/SIMT** hardveren.
- Megismerni a **warpokra bontást**: hogyan lesz a blokk szálaiból warp.
- Megérteni a **vezérlési divergencia** (control divergence) okát, költségét és mérését.
- Megtanulni a **párhuzamos redukció** mintáját, és lépésről lépésre optimalizálni.
- Kitekintés: warp-szintű primitívek (`__shfl_down_sync`), könyvtárak (CUB, Thrust).

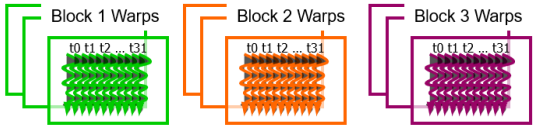
Megjegyzés

SIMD (Single Instruction, Multiple Data): egyetlen utasítás több adaton fut egyszerre. Ez adja a GPU-k nagy számítási teljesítményét: a vezérlés költsége sok számolóegység között oszlik meg.

Tartalom

1. Warpok és SIMT végrehajtás
2. Vezérlési divergencia
3. Párhuzamos redukció
4. Haladó optimalizációk
5. Összefoglalás

A warp mint ütemezési egység



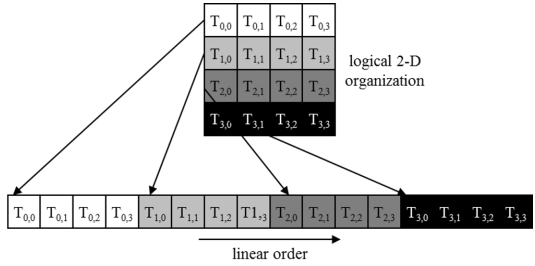
- Minden blokk **32 szálas warpokra** oszlik (a kódban: `warpSize`).
- Ez *megvalósítási részlet*, nem a CUDA programozási modell része – mégis erősen befolyásolja a teljesítményt.
- A warp az ütemezés alapegysége az **SM**-en (Streaming Multiprocessor).
- Egy warp szálai ugyanazt az utasítást hajtják végre, különböző adatokon.

Megjegyzés

SM: a GPU feldolgozó egysége, ahol a szálak ténylegesen futnak. Egy blokk mindig *egyetlen* SM-re kerül, és ott is marad, amíg be nem fejeződik.

Warpok többdimenziós blokkokban

- A többdimenziós blokkot először **lineáris sorrendbe** rendezzük (sorfolytonosan).
- A sorrend: először `x`, majd `y`, végül `z`.
- Lineáris index (3D):
$$tid = \mathtt{z} \cdot D_y D_x + \mathtt{y} \cdot D_x + \mathtt{x}$$
- Ebből: $warp = \lfloor tid/32 \rfloor$, $lane = tid \bmod 32$



Példa

16 × 16-os blokkban a 0. warp a `threadIdx.y = 0` és 1 sorok 32 szála, az 1. warp a 2. és 3. sor, és így tovább: összesen 8 warp.

Feladat: hány warp?

- Legyen `blockDim = (20, 10, 1)`.
- Hány warp jön létre? Mely szálak alkotják a 0. warpot? Hány aktív lane van az utolsó warpban?

Feladat: hány warp?

- Legyen `blockDim = (20, 10, 1)`.
- Hány warp jön létre? Mely szálak alkotják a 0. warpot? Hány aktív lane van az utolsó warpban?

Megoldás

- $20 \cdot 10 = 200$ szál $\Rightarrow \lceil 200/32 \rceil = \mathbf{7}$ warp (6 teli + 1 részleges).
- 0. warp: $tid = 0..31 \Rightarrow$ a y=0 sor mind a 20 szála + a y=1 sor első 12 szála.
- Utolsó warp: $tid = 192..199 \Rightarrow$ csak **8** aktív lane, 24 lane kihasználatlan.

▶ Haladó

A kihasználtság itt $200/(7 \cdot 32) \approx 89\%$. Egy 32×8-as blokk ugyanígy 256 szál, de 100%-os lane-kihasználtsággal.

Blokk-partícionálás a linearizálás után

- A lineáris szálindexek warpon belül egymás után növekednek.
- A **0. warp** a **0. szállal** kezdődik.
- A partícionálási séma minden eszközön azonos.
- A **warpok egymáshoz képesti végrehajtási sorrendjére** soha ne építsünk – az ütemező tetszőleges sorrendben futtatja őket.
- Ha a szálak között függőség van, szinkronizáljunk: `__syncthreads()`.

Megjegyzés

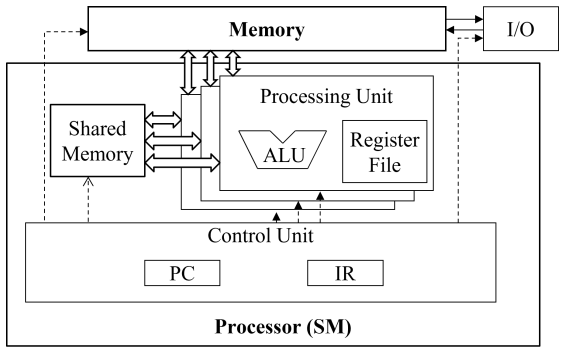
`__syncthreads()`: **blokkszintű** gát (barrier). A blokk *összes* szála megvárja egymást, és a gát előtti `__shared__`/globális írások a gát után a blokk minden szála számára láthatók.

▶ Haladó

`__syncthreads()` csak akkor helyes, ha a blokk *minden* szála eléri. Divergens ágba téve holtpontot vagy definiálatlan viselkedést okozhat.

SM-ek mint SIMD processzorok

- Az utasítások lekérése, dekódolása és ütemezése **közös** több feldolgozó egység számára.
- Ez csökkenti a vezérlés költségét (terület, energia) – több tranzisztor jut számolásra.
- Az SM fő részei:
  - **ALU**-k / feldolgozó egységek,
  - **regiszterfájl**,
  - **megosztott memória** (shared memory),
  - **vezérlőegység**: **PC** (Program Counter), **IR** (Instruction Register).



SIMD vagy SIMT? – Független szálütemezés

Klasszikus SIMD (CPU: AVX, NEON)

- Egy utasítás egy *vektorregiszteren* dolgozik.
- A programozó/fordító vektorizál.
- Elágazás: kézi maszkolás.

SIMT (NVIDIA GPU)

- Szálanként írt, skalár kód.
- Saját regiszterek, saját címszámítás.
- Elágazás: a hardver maszkol (aktív maszk).

▶ Haladó: Volta (sm_70) óta

Independent Thread Scheduling: minden szálnak saját PC-je és hívási verme van. A divergens ágak *átlapolódhatnak*, a szálak nem garantáltan konvergálnak újra. A régi, implicit „warp-szinkron" trükkök helyett használjuk a `__syncwarp()` és a `*_sync(mask, ...)` primitíveket!

Késleltetés elrejtése: miért kell sok warp?

- Egy globális memóriaolvasás **több száz órajelciklusig** tart.
- Amíg egy warp adatra vár, az ütemező egy másik, **futásra kész** warpot választ.
- A váltás **nulla költségű**: minden rezidens warp regiszterei az SM-en vannak.
- Minél több rezidens warp van, annál jobban elrejthető a késleltetés.

Occupancy (kihasználtság)

$$\text{Occupancy} = \frac{\text{aktív warpok száma egy SM-en}}{\text{maximálisan rezidens warpok száma}}$$

Korlátozza: blokkonkénti szálszám, szálankénti regiszterhasználat, blokkonkénti shared memória.

SIMD végrehajtás warp szinten

- Egy warp összes szála egy adott időpontban ugyanazt az utasítást hajtja végre.
- Ez akkor hatékony, ha minden szál **ugyanazt a vezérlési utat** követi:
  - az `if`-ágak minden szálra ugyanúgy döntenek,
  - a ciklusok minden szálra ugyanannyi iterációt futnak.

Megjegyzés – miért pont 32?

- **Memória**: 32×4 bájt $= 128$ bájt $=$ egy L1 cache-sor. Egy warp `float`-olvasása ideális esetben egyetlen tranzakció.
- **Hardverszélesség**: kompromisszum az ALU-szélesség és a vezérlés költsége között.
- **Ütemezés**: kevesebb, de szélesebb ütemezési egység – olcsóbb késleltetés-elrejtés.

Megjegyzés

Ha a szálak eltérő ágakat választanak, **vezérlési divergencia** lép fel, ami lassítja a futást.

Tartalom

1. Warpok és SIMT végrehajtás
2. Vezérlési divergencia
3. Párhuzamos redukció
4. Haladó optimalizációk
5. Összefoglalás

Vezérlési divergencia

- Akkor lép fel, ha egy warp szálai **különböző vezérlési utakat** választanak, például:
  - egyes szálak az **if**, mások az **else** ágon futnak;
  - egyes szálak több ciklus-iterációt hajtanak végre, mint mások.
- Az eltérő utak **egymás után** (sorosan) hajtódnak végre, nem párhuzamosan:
  - a warp szálai által bejárt utakat a hardver egyenként futtatja le;
  - egy-egy út futtatásakor az azt bejáró szálak párhuzamosan dolgoznak, a többi lane **maszkolva** van;
  - beágyazott vezérlési szerkezeteknél az utak száma gyorsan nőhet.
- Következmény: csökken a párhuzamos hatékonyság.

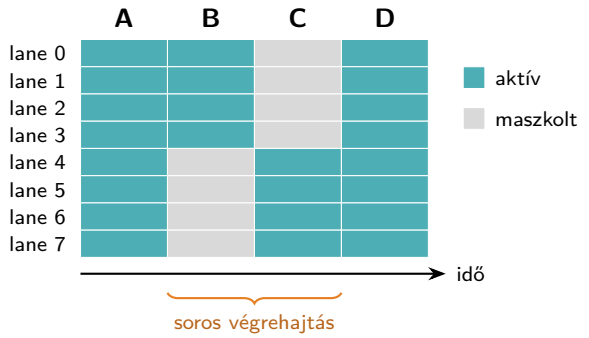
Megjegyzés

A divergencia **csak warpon belül** értelmezett: ha két *különböző* warp más utat választ, az nem okoz lassulást.

Divergencia szemléltetése: aktív maszk

```
A();                    // minden szál
if (threadIdx.x < 4)
    B();                // lane 0..3
else
    C();                // lane 4..7
D();                    // újra együtt
```

Az egyszerűség kedvéért 8 lane-es warpot rajzolunk.



Példa: divergens és nem divergens elágazás

- Divergencia akkor léphet fel, ha az elágazás vagy a ciklus feltétele a **szálindex függvénye**.

Divergens eset:

- `if (threadIdx.x > 2) { ... }`
- Két vezérlési út jön létre a blokk szálai számára.
- A döntés granularitása kisebb a warpméretnél: a 0., 1. és 2. szál más utat követ, mint a 0. warp többi szála.

Nem divergens eset:

- `if (blockIdx.x > 2) { ... }`
- A döntés granularitása a teljes blokk: egy warp minden szála ugyanazt az utat követi.
- `if (threadIdx.x / warpSize > 2)` szintén nem divergál: a döntés **warp-granularitású**.

▶ Haladó

Ökölszabály: ha a feltétel csak `blockIdx`-től, `threadIdx.x / 32`-től vagy uniform kernelparamétertől függ, nincs divergencia.

Divergencia csökkentése a gyakorlatban

1. **Warp-granularitású döntések**: a munkát úgy osszuk ki, hogy egy warp szálai egyformán döntsenek.
2. **Adatok átrendezése**: hasonló munkájú elemek kerüljenek egy warpba (pl. rendezés típus vagy hossz szerint).
3. **Elágazásmentes kód**: `fminf`, `fmaxf`, ternáris operátor (*select*) – a fordító predikált kódot generál.
4. **Munka tömörítése** (compaction): az aktív elemeket gyűjtsük egy sorba, és csak azokat dolgozzuk fel.

```
// Elágazással:                    // Elágazás nélkül:
if (x < 0.0f) y = 0.0f;            y = fmaxf(x, 0.0f);   // ReLU
else          y = x;
```

Divergencia mérése: Nsight Compute

- A divergencia nem csak elméleti kérdés – **profilozóval mérhető**.
- **Nsight Compute** (`ncu`) hasznos szekciói:
  - *Source Counters*: divergens elágazások, forráskódsorokhoz rendelve;
  - *Warp State Statistics*: miért állnak a warpok (memória, szinkronizáció, . . .).
- Kulcsmetrika: átlagosan hány szál aktív egy warp-utasításnál (ideális: 32).

```
ncu --section SourceCounters --section WarpStateStats ./reduction
ncu --metrics smsp__thread_inst_executed_per_inst_executed.ratio ./reduction
```

▶ Haladó

Warp Execution Efficiency ≈ (aktív szálak átlaga utasításonként) / 32. A naiv és a javított redukciós kernel között ez az érték jól látható különbséget mutat.

A vezérlési divergencia teljesítményhatása

- A **határellenőrzések** elengedhetetlenek, ha a méret nem osztható a csempemérettel; a **csempézett mátrixszorzás** sok ilyet tartalmaz.
- Elsőre lassítónak tűnnek, de a hatásuk **mérsékelt** – ezt most számszerűen is megnézzük.

Példa – mátrixcsempék betöltése a p-edik fázisban

```
// M csempe: sor = Row, oszlop = p*TILE_WIDTH + tx
if (Row < Width && p*TILE_WIDTH + tx < Width)
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH + tx];
else
    ds_M[ty][tx] = 0.0f;      // kívül eső elem: semleges 0

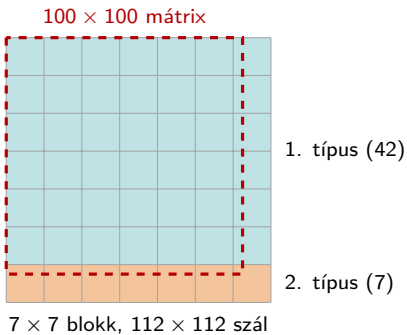
// N csempe: sor = p*TILE_WIDTH + ty, oszlop = Col
if (p*TILE_WIDTH + ty < Width && Col < Width)
    ds_N[ty][tx] = N[(p*TILE_WIDTH + ty)*Width + Col];
else
    ds_N[ty][tx] = 0.0f;
```

Kétféle blokk az M mátrix betöltésénél

1. **1. típus:** a blokk minden sora a mátrixon belül van (felső 6 blokksor). Csak az *utolsó fázisban* lóg ki a csempe oszlopirányban.
2. **2. típus:** a blokk sorai részben a mátrixon kívül esnek (alsó blokksor).

Emlékeztető

TILE_WIDTH: a csempe szélessége, itt 16, azaz 16×16 elemes csempék és 16×16 szálas blokkok.



A divergencia hatásának elemzése – kiinduló adatok

- **Csempe- és blokkméret**: 16×16.
- Blokkonként $256/32 = 8$ **warp**; minden warp 2 teljes sort fed le.
- Négyzetes mátrix: 100×100.
- Fázisok száma: $\lceil 100/16 \rceil = 7$.
- Blokkok száma: $7 \times 7 = 49$.
- Összes **warp-fázis**: $49 \cdot 8 \cdot 7 = 2744$.

Megjegyzés

Warp-fázis: egy warp egy fázisbeli betöltése. Azt számoljuk meg, hány warp-fázisban divergál a betöltés feltétele.

Divergencia az M csempék betöltésében – 1. típus

- $6 \times 7 = 42$ darab 1. típusú blokk $\Rightarrow 42 \cdot 8 = 336$ warp.
- Mind a 7 fázison végigmennek: $336 \cdot 7 = 2352$ **warp-fázis**.
- Az első 6 fázisban minden elem érvényes $\Rightarrow$ nincs divergencia.
- A 7. (utolsó) fázisban az oszlopok 96..111: csak `tx` $= 0..3$ érvényes.
  Minden warp tartalmaz érvényes és érvénytelen szálat $\Rightarrow$ **minden warp divergál**.
- Divergens: 336 warp-fázis a 2352-ből.

Divergencia az M csempék betöltésében – 2. típus

- 7 darab 2. típusú blokk (alsó blokksor, sorok: 96..111).
- Összesen $7 \cdot 8 = 56$ warp $\Rightarrow 56 \cdot 7 = 392$ warp-fázis.
- Érvényes sorok: 96..99 $\Rightarrow$ `ty` $= 0..3 \Rightarrow$ csak a **0. és 1. warp** tartalmaz érvényes sort.
  - Ez a két warp csak az utolsó fázisban divergál (ugyanúgy, mint az 1. típusnál).
- A többi 6 warp **teljesen** az érvénytelen tartományban van: mindig az `else` ágat futtatja $\Rightarrow$ *nincs* divergencia.
- Divergens: $2 \cdot 7 = 14$ warp-fázis a 392-ből.

A vezérlési divergencia összesített hatása

- 1. típus: 336/2352 divergens warp-fázis.
- 2. típus: 14/392 divergens warp-fázis.
- Összesen: 350/2744 $\approx$ **12,8**% divergens warp-fázis.
- Mivel csak a rövid betöltő kód divergál, a valós teljesítményveszteség **ennél jóval kisebb**.

Méretfüggés

Mátrixméret	Fázisok	Divergens warp-fázis	Arány
100×100	7	350/2744	$\approx$ 12,8%
1000×1000	63	31 500/2 000 376	$\approx$ 1,6%

További megjegyzések

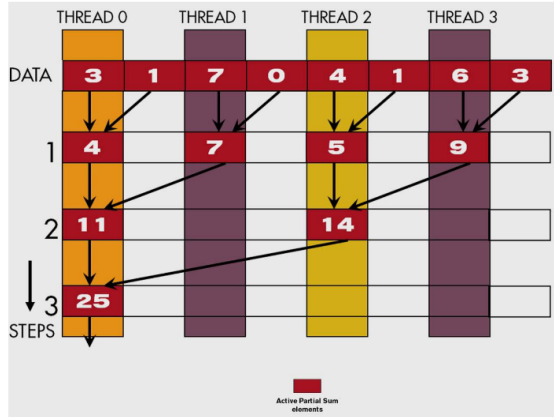
- Az N mátrix csempéinek betöltése hasonló, de kissé eltérő divergenciamintázatot mutat.
- A hatás **méretfüggő**: nagyobb mátrixoknál a divergens arány még kisebb.
- Nagy bemeneteknél a határfeltételek ellenőrzése alig lassít.
- Ne féljünk **if** feltételeket használni a helyes működés érdekében!
- A sok elágazás nem feltétlenül jelent nagy divergenciaterhelést.
- Vannak azonban **természetüknél fogva divergens** algoritmusok $\rightarrow$ **redukció**.

Tartalom

1. Warpok és SIMT végrehajtás
2. Vezérlési divergencia
3. Párhuzamos redukció
4. Haladó optimalizációk
5. Összefoglalás

Redukciós algoritmus

- A **párhuzamos redukció** az egyik legfontosabb párhuzamos algoritmikus séma.
- Cél: **hatékony redukciós kernel** írása.
- Elemzési szempontok:
  - **munkahatékonyság**,
  - **erőforrás-használat**,
  - **vezérlési divergencia**.



„Felosztás és összegzés” stratégia

- Nagy adathalmaz feldolgozásakor gyakori módszer:
  1. az adatokat kisebb darabokra (*chunk*) osztjuk;
  2. minden szál feldolgoz egy darabot;
  3. a részeredményeket egy **redukciós fával** (reduction tree) összegezzük.
- Feltétel: a művelet legyen **asszociatív**; ha az elemek sorrendjét is átrendezzük, **kommutatív** is.
- Példák: MapReduce, GPU-s redukciók (összeg, maximum, . . .).

▶ Haladó

A lebegőpontos összeadás **nem asszociatív**! A párhuzamos összeg ezért bitre pontosan eltérhet a szekvenciálistól (lásd később).

Mi az a redukciós művelet?

- Több bemeneti értékből egyetlen összesítő értéket állít elő.
- Lehet például: maximum, minimum, összeg, szorzat, logikai ÉS/VAGY.
- Kell, hogy legyen **egységelem** (identitáselem):
  - összeghez 0, szorzathoz 1, max-hoz $-\infty$, min-hez $+\infty$.
- A művelet gyakran *kollektív*: több szál együttműködésével történik.

▶ Haladó: összetett redukciók

argmax: (érték, index) párok; **átlag**: (Σ, n) párok; **szórás**: Welford/Chan-féle összevonás; **logsumexp** (softmax): (m, s) párok, ahol m a maximum és s az átskálázott összeg.

Szekvenciális redukció – $O(N)$

Inicializáljuk az eredményt a művelet egységelemével:

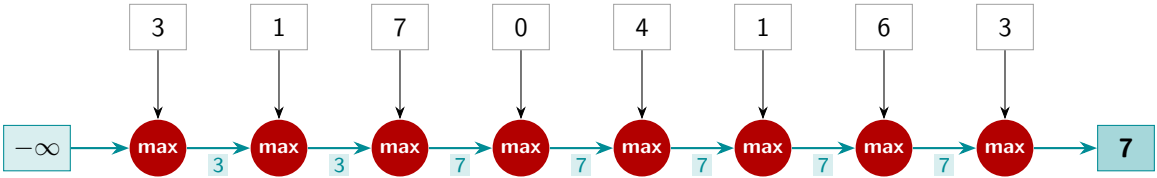
- maximumnál a lehető legkisebb érték,
- minimumnál a lehető legnagyobb érték,
- összegnél 0, szorzatnál 1.

Ezután haladjunk végig a bemeneten, és vonjuk össze az aktuális eredményt a következő elemmel.

- N bemenet $\Rightarrow$ N művelet.
- Minden elemet pontosan egyszer érintünk: $O(N)$.
- Számítási szempontból hatékony.

```
// max egységeleme: -végtelen
float result = -INFINITY;
for (int i = 0; i < N; ++i)
    result = fmaxf(result, input[i]);
```

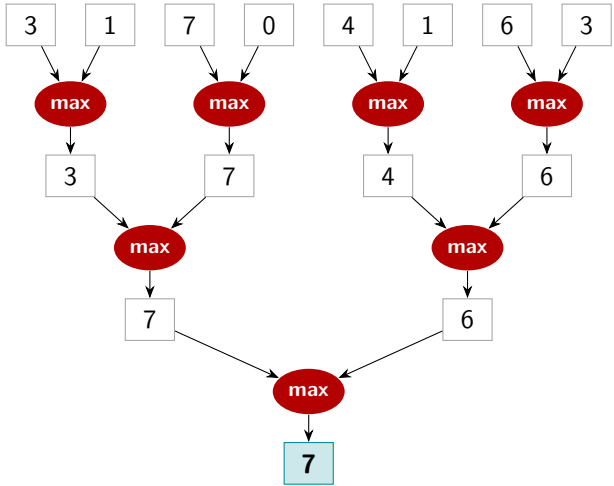
Szekvenciális redukció – példa (max)



akkumulátor (futó eredmény)

8 bemenet ⇒ 8 művelet, 8 *egymásra épülő* lépés: nincs párhuzamosság.

Párhuzamos redukciós fa



1. lépés

4 művelet

2. lépés

2 művelet

3. lépés

1 művelet

8 bemenet $\Rightarrow 4 + 2 + 1 = 7$ művelet, de csak $\log_2 8 = 3$ lépés.

A párhuzamos redukciós fa elemzése

N bemeneti érték esetén:

- Minden lépésben a szálak fele (az előzőhöz képest) két elemet von össze.
- Műveletek száma: $\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots + 1 = N - 1$.
- Lépések száma: $\log_2 N$ – pl. $N = 1\,000\,000$ esetén mindössze 20 lépés.
- Feltéve, hogy elegendő végrehajtási erőforrás áll rendelkezésre.
- Átlagos párhuzamosság: $\frac{N-1}{\log_2 N}$; $N = 1\,000\,000$ esetén $\approx 50\,000$.
- A csúcsidejű erőforrásigény viszont $N/2 = 500\,000$ szál – ez **nem erőforrás-hatékony**.
- **Munkahatékony** (ugyanannyi művelet, mint a szekvenciális változatban), de az **erőforrás-hatékonyság** korlátozott.

Párhuzamos redukció CUDA-ban – alapötlet

- Egy blokk 2·`blockDim.x` bemeneti elemet dolgoz fel.
- Minden szál két értéket ad össze egy lépésben.
- Az aktív szálak száma lépésenként feleződik.
- A redukció mélysége $\log_2(2 \cdot \texttt{blockDim.x})$ lépés.
- A részösszegek a **megosztott memóriában** (`__shared__`) tárolódnak.
- A blokk eredménye az első elemben (`partialSum[0]`) keletkezik.

Megjegyzés

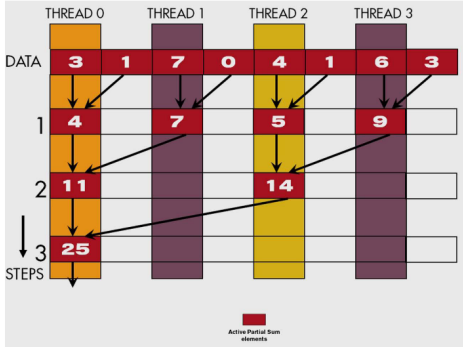
A megosztott memória kb. egy nagyságrenddel gyorsabb a globális memóriánál, így a köztes eredmények nem terhelik a DRAM-ot.

Egyszerű (naiv) redukciós kernel

```
__global__ void reduceNaive(const float *input,
                            float *output) {
  __shared__ float partialSum[2*BLOCK_SIZE];
  unsigned int t = threadIdx.x;
  unsigned int start = 2*blockIdx.x*blockDim.x;

  // minden szál két elemet tölt be
  partialSum[t] = input[start + t];
  partialSum[blockDim.x + t] =
                  input[start + blockDim.x + t];

  for (unsigned int stride = 1;
       stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    if (t % stride == 0)
      partialSum[2*t] += partialSum[2*t + stride];
  }
  if (t == 0) output[blockIdx.x] = partialSum[0];
}
```



Miért kell a `__syncthreads()`?

- Blokkszintű gát: minden szálnak be kell fejeznie az adott lépést, mielőtt a következő elkezdődik.
- Nélküle **adatverseny** (race condition) alakulna ki: egy szál olyan részösszeget olvashatna, amit egy másik warp még nem írt meg.
- Garantálja, hogy a következő iteráció a `partialSum[]` friss értékeit látja.
- A ciklus *elején* álló gát az első iterációban a **betöltéseket** is lezárja.

▶ Haladó

Ha a még aktív szálak mind egyetlen warpban vannak (`stride < 32`), elegendő lenne a `__syncwarp()`, ami olcsóbb. A blokkszintű gát viszont mindig helyes.

A globális eredmény előállítása

- A blokk 0. szála elmenti a blokkösszeget: `output[blockIdx.x] = partialSum[0]`.
- A blokkonkénti részösszegeket ezután:
  - egy újabb kernelhívással tovább redukáljuk (rekurzívan), vagy
  - ha kevés blokk van, a CPU-n adjuk össze.
- Alternatíva: **atomikus művelettel** egy kernelben összegzünk:

```
if (t == 0) atomicAdd(output, partialSum[0]);   // *output előtte 0-ra állítva!
```

▶ Haladó

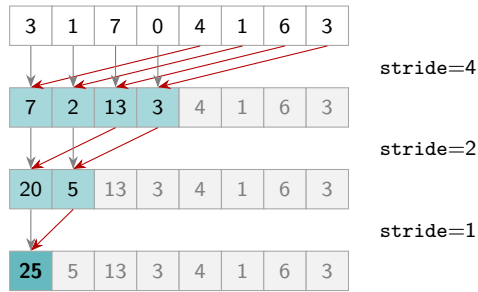
Az `atomicAdd` sorrendje futásonként változhat ⇒ `float` esetén az eredmény **nem determinisztikus** (bitre). Reprodukálható eredményhez fix fát vagy kétlépéses redukciót használjunk.

A naiv kernel problémái

- Minden iterációban a warpok **két vezérlési utat** követnek sorban:
  - az összeadást végző szálak útját,
  - és a többiekét.
- Az összeadást nem végző szálak is **foglalják a végrehajtási erőforrásokat**.
- Az első lépés után már csak a szálak fele dolgozik: minden páratlan indexű szál inaktív.
- Az 5. lépés után (`stride` ≥ 32) egy warpban legfeljebb **1 szál** dolgozik:
  - egyes warpokban pontosan egy (divergens, $1/32$ kihasználtság),
  - a többi warp minden szála elbukik az `if` teszten (nincs divergencia, de pazarló).
- Ez a 32, 64, 128, 256, 512, 1024 lépésközöknél végig így marad, amíg a blokk összes warpja be nem fejeződik.

Indexelés és tömörítés (compaction)

- Ötlet: az aktív szálak legyenek **egymás mellett**.
- Az **asszociativitás és kommutativitás** miatt az összeadások sorrendje szabadon választható.
- Minden lépésben a részösszegeket a tömb **elejére** gyűjtjük.
- Így az aktív szálak mindig folytonos indexeken dolgoznak.



Javított redukciós kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2) {
    __syncthreads();
    if (t < stride)                          // aktív szálak: 0 .. stride-1
        partialSum[t] += partialSum[t + stride];
}
if (t == 0) output[blockIdx.x] = partialSum[0];
```

- A betöltés ugyanaz, mint a naiv kernelben; csak a ciklus változik.
- Az aktív szálak mindig a **0..stride-1** tartományban vannak.
- Amíg `stride` ≥ 32, minden warp vagy **teljesen aktív**, vagy **teljesen inaktív**.

Előny

Az első lépésekben nincs divergencia, a nem dolgozó warpok pedig **teljes egészükben** kimaradnak – az ütemező nem pazarol rájuk utasítás-kiadást.

Gyors elemzés

- 1024 szálas blokk, 2048 elem, `stride` $= 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1$ (11 lépés).
- Az első **6** lépésben (`stride` ≥ 32) **nincs divergencia**: minden warp vagy teljesen aktív, vagy teljesen inaktív.
- Az utolsó **5** lépésben (`stride` ≤ 16) csak **egyetlen warp** (a 0.) divergál.
- A naiv változatban ezzel szemben a 2–5. lépésben *minden* warp divergál, utána pedig az aktív warpok 1/32-es kihasználtsággal futnak (191 vs. 5 divergens warp-lépés).
- Az algoritmus így **erőforrás-hatékonyabb**, továbbra is munkahatékony ($N - 1$ művelet), és kisebb a divergencia költsége.

▶ Haladó

Az utolsó 5 lépés még mindig pazarló (1 warp, egyre kevesebb aktív szál, minden lépés után blokkszintű gát). Megoldás: **warp-szintű redukció** shuffle utasításokkal.

Tartalom

1. Warpok és SIMT végrehajtás
2. Vezérlési divergencia
3. Párhuzamos redukció
4. Haladó optimalizációk
5. Összefoglalás

Warp-szintű redukció shuffle utasításokkal

- `__shfl_down_sync(mask, v, d)`: a lane a *lane+d*-edik szál `v` értékét kapja meg – közvetlenül **regiszterből**, shared memória és `__syncthreads()` nélkül.

```
__inline__ __device__ float warpReduceSum(float v) {
    for (int offset = warpSize / 2; offset > 0; offset /= 2)
        v += __shfl_down_sync(0xffffffff, v, offset);
    return v;                                  // az eredmény a 0. lane-ben
}

__inline__ __device__ float blockReduceSum(float v) {
    __shared__ float warpSums[32];             // max. 1024/32 = 32 warp
    int lane = threadIdx.x % warpSize, wid = threadIdx.x / warpSize;
    v = warpReduceSum(v);                      // 1) warponként
    if (lane == 0) warpSums[wid] = v;
    __syncthreads();                           // 2) egyetlen blokkszintű gát
    int nWarps = (blockDim.x + warpSize - 1) / warpSize;
    v = (threadIdx.x < nWarps) ? warpSums[lane] : 0.0f;
    if (wid == 0) v = warpReduceSum(v);        // 3) a warp-összegek redukciója
    return v;                                  // eredmény: threadIdx.x == 0
}
```

Szálösszevonás (thread coarsening) és grid-stride ciklus

```
__global__ void reduceSum(const float *__restrict__ in, float *out, int n) {
    float sum = 0.0f;
    // 1) szekvenciális rész: minden szál sok elemet összegez (olcsó!)
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n;
         i += gridDim.x * blockDim.x)
        sum += in[i];                          // coalesced olvasás
    // 2) párhuzamos rész: blokkon belüli fa (warp shuffle)
    sum = blockReduceSum(sum);
    // 3) blokkonként egy atomikus művelet
    if (threadIdx.x == 0) atomicAdd(out, sum);
}
// Indítás: annyi blokk, hogy minden SM-et kitöltsön, pl. 4-8 blokk/SM
```

- Tetszőleges n-re működik (nincs feltevés a méretről).
- Kevesebb szál, kevesebb szinkronizáció $\Rightarrow$ a futásidő a **memória-sávszélességhez** közelít.

Buktató: lebegőpontos pontosság

- A lebegőpontos összeadás **nem asszociatív**: $(a + b) + c \neq a + (b + c)$.

```
float a = 1.0e8f, b = 1.0f;
(a + b) - a   // == 0.0f   (1e8 körül a float lépésköze 8)
(a - a) + b   // == 1.0f
```

- A párhuzamos fa más sorrendben ad össze $\Rightarrow$ **más eredmény**, mint a CPU-s ciklus. Ez nem hiba!
- A fa alakú összegzés hibája jellemzően **kisebb**: $O(\log N)$, szemben a szekvenciális $O(N)$-nel.
- Tesztelésnél relatív hibakorláttal hasonlítsunk, ne ==-vel.

▶ Haladó

Pontosabb összegzés: `double` akkumulátor, **Kahan-összegzés** (kompenzált).
Reprodukálhatóság: atomikus műveletek helyett fix redukciós fa.

A gyakorlatban: használjunk könyvtárat!

- Éles kódban a redukciót ritkán írjuk meg kézzel – a könyvtárak architektúránként hangoltak.

```
// Thrust (magas szintű, STL-szerű)
float s = thrust::reduce(thrust::device, d_in, d_in + n, 0.0f);

// CUB  eszközszintű redukció (kétlépéses: előbb a temp tárigény)
void *tmp = nullptr; size_t bytes = 0;
cub::DeviceReduce::Sum(tmp, bytes, d_in, d_out, n);
cudaMalloc(&tmp, bytes);
cub::DeviceReduce::Sum(tmp, bytes, d_in, d_out, n);

// CUB  blokkszintű építőelem saját kernelben
using BlockReduce = cub::BlockReduce<float, 256>;
__shared__ typename BlockReduce::TempStorage temp;
float blockSum = BlockReduce(temp).Sum(threadValue);
```

Tartalom

1. Warpok és SIMT végrehajtás
2. Vezérlési divergencia
3. Párhuzamos redukció
4. Haladó optimalizációk
5. Összefoglalás

Összegzés

- A **warp** (32 szál) a végrehajtás és ütemezés alapegysége; a szálak SIMT módon futnak.
- A **vezérlési divergencia** warpon belül sorosítja az ágakat. A határellenőrzések költsége azonban nagy bemenetnél kicsi.
- A **redukció** az egyik legfontosabb párhuzamos minta: $N - 1$ művelet, $\log_2 N$ lépés.
- A **szekvenciális címzés** (kompakt indexelés) megszünteti a divergencia nagy részét és a bank-konfliktusokat.
- A **shuffle**, a **thread coarsening** és a könyvtárak (CUB/Thrust) közel sávszélesség-korlátos teljesítményt adnak.

Továbbfejlesztési irányok

Prefix-összeg (scan) és stream compaction; hisztogram atomikus műveletekkel; szegmentált redukció; memória-coalescing részletesen.

Gyakorló feladatok

1. `blockDim = (8, 8, 4)`: hány warp van? Mely `(x,y,z)` szálak alkotják a 3. warpot?
2. Divergál-e: `if (threadIdx.x % 64 < 32)`? És `if (threadIdx.x % 2 == 0)`?
3. Számoljuk ki a divergens warp-fázisok arányát 200×200-as mátrixra, 16×16-os csempével!
4. Írjunk **max + argmax** redukciót (érték és index párosával)!
5. Mérjük meg Nsight Compute-tal a naiv és a javított kernelt: futásidő, sávszélesség (GB/s), aktív szál/utasítás.
6. Hasonlítsuk össze a saját kernelt a `cub::DeviceReduce::Sum` függvénnyel $N = 2^{26}$ elemre!

Ajánlott irodalom

- W.-m. Hwu, D. B. Kirk, I. El Hajj: *Programming Massively Parallel Processors*, 4. kiadás, Morgan Kaufmann, 2022. – 4. fejezet (architektúra, ütemezés, divergencia) és 10. fejezet (redukció).
- M. Harris: *Optimizing Parallel Reduction in CUDA*, NVIDIA, 2007.
- NVIDIA: *CUDA C++ Programming Guide* – „SIMT Architecture", „Warp Shuffle Functions".
- NVIDIA: *CUDA C++ Best Practices Guide* – „Branching and Divergence".
- J. Luitjens: *Faster Parallel Reductions on Kepler*, NVIDIA Developer Blog, 2014.
- Y. Lin, V. Grover: *Using CUDA Warp-Level Primitives*, NVIDIA Developer Blog, 2018.
- NVIDIA: *Tesla V100 GPU Architecture Whitepaper*, 2017 – Independent Thread Scheduling.
- CCCL dokumentáció (Thrust, CUB): `https://nvidia.github.io/cccl/`