Grafikus kártyák programozása

Optimalizálás

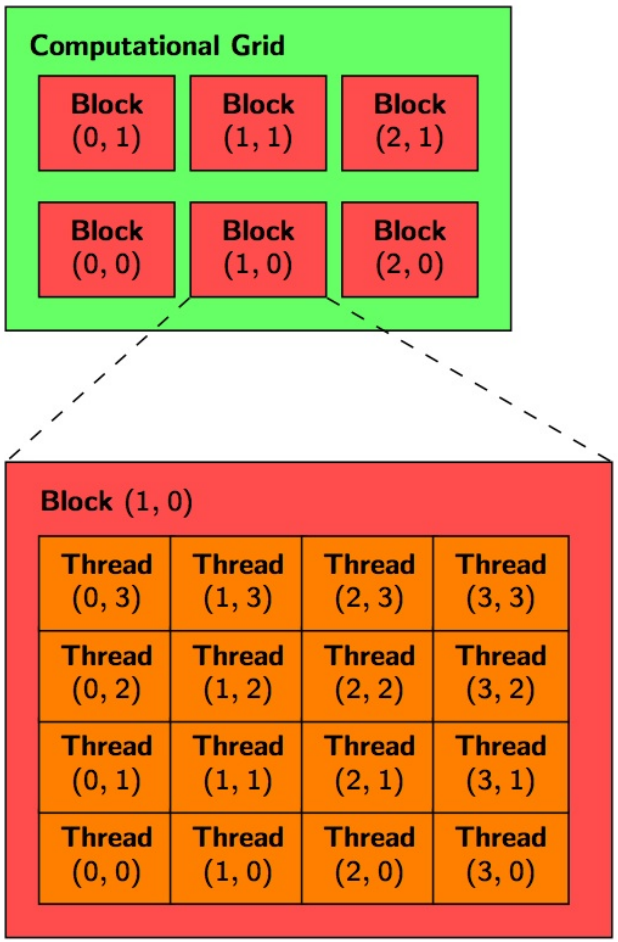
Tóth Gyula

MTA-SZFKI

MTA-RMKI, Budapest, 2011. július 8.

Tartalom

A CUDA futtatási modell



A számítási háló (**Computational grid**)

1. A párhuzamosan futtatandó program a mag (**kernel**)
2. A magot szálak (**thread**) futtatják
3. A szálak 1,2 v. 3 dimenziós logikai csoportja a blokk (**block**)
4. A blokkok 1 v. 2 dimenziós logikai csoporta a teljes számítási háló (**grid**)

Adatpárhuzamosítás

Egyértelmű háló $\Leftrightarrow$ adathalmaz megfeleltetés az egyéni szálazonosítókon keresztül

Implementáció

Feladat: az adathalmaz párhuzamosítása (**grid**) + a számítási feladat párhuzamosítása (**kernel**)

Make it work. Make it right. Make it fast.

"Premature optimization is the root of all evil."

A párhuzamos optimalizálás területei

- Az implementáció optimalizálása az architekturára
- Az optimális implementáció optimalizálása

Megéri?

- **A feladatmegoldás ideje (τ) = programfejlesztési idő (τ_p) + futtatási idő (τ_f)**
- Hibajavítási (**debug**) idő: $\tau_d^{\text{soros}} << \tau_d^{\text{párhuzamos}}$
- Murphy: $\tau_p \approx \tau_d \Rightarrow$ Előfordulhat, hogy $\tau^{\text{soros}} < \tau^{\text{párhuzamos}}$

Make it work. - A működő párhuzamos kódig

$n * m$-es mátrixok összeadása

CPU függvény

```
void add_matrix(float *a,float *b,float *c,int
n,int m){

int idx;

for (idx=0; idx<n*m; idx++)
c[idx]=a[idx]+b[idx];

}
```

- Lineáris tömbkezelés:
  a[n][m] 2D tömb NEM folytonos
- Egyszerű soros ciklus
- CPU *host* memórián dolgozik
  $\Rightarrow$ A főprogramból közvetlenül hívható

GPU mag

```
__global__
void add_matrix(float *a,float *b,float *c,int n,int m){

int whoami=blockDim.x*blockIdx.x+threadIdx.x;

if (whoami<n*m)
c[whoami]=a[whoami]+b[whoami];

}
```

- Eltűnt a *for* ciklus,
  minden szál egyetlen összeadást végez
  $whoami = 0 \ldots gridDim.x * blockDim.x - 1$
- Lineáris tömbkezelés, 1D háló: Ha
  $blockDim.x * gridDim.x \geq n * m$
  $\Rightarrow$ minden elemet lefedtünk!
- GPU *device* memórián dolgozik

Make it work. - A működő párhuzamos kódig

$n * m$-es mátrixok összeadása

Hívás a CPU-ról

```
__device__
float *a_gpu,*b_gpu,*c_gpu;

main(){

float *a,*b,*c;

// ...

cudaMemcpy(a_gpu,a,N*M*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(b_gpu,b,N*M*sizeof(float),cudaMemcpyHostToDevice);

dim3 threads(THREADS);
dim3 grid(N*M/THREADS+1);

add_matrix<<<grid,threads>>>(a_gpu,b_gpu,c_gpu,N,M);

cudaMemcpy(c,c_gpu,N*M*sizeof(float),cudaMemcpyDeviceToHost);

// ...

}
```

GPU mag

```
__global__
void add_matrix(float *a,float *b,float *c,int
n,int m){

int whoami=blockDim.x*blockIdx.x+threadIdx.x;

if (whoami<n*m)
c[whoami]=a[whoami]+b[whoami];

}
```

Folyamat

- Adatok felmásolása a GPU-ra
- A mag futtatása megadott hálón
- Az eredmény másolása a CPU-ra

Make it right. - A helyes eredmény

Az eredmény helyessége

- Informatikai helyesség:
  helyes kernel műveletsor (soros debug) & helyes párhuzamosítás (!)
- Pontosság:
  Az informatikailag helyesen futó program eredménye megfelel a kívánalmaknak?

Általános GPU hibák

- Informatikai hibák:
  Helytelen párhuzamos adatkezelés, szinkronizációs hiba, rossz adathalmaz lefedés
- Pontosság:
  A GPU **nativ** műveleteinek ismerete (IEEE-754 kivételek)

Figyelem!

A GPU gyors 3D ábrázolásra készült, nem tudományos szuperszámítógépnek!

Make it fast. - Optimalizálás

Memóriahasználat

- global memory (**coalesced memory access**):
  A (**lassú**) globális memória írása/olvasása **nagyságrendekkel** felgyorsítható.
- multiprocessor cache (**shared memory**):
  kisebb ($\sim 16kB$), de **gyors** memória, egy blokk számára látható

Egyéb

- Ciklus kifejtés (loop unrolling): $\tau_{loop+arithmetic} > N \times \tau_{arithmetic}$
- Beépített adattípusok - float2, float3, float4, double2, stb.
- GPU nativ függvények - minél pontosabb, annál lassabb...
- Algoritmus csere - hatékonyabb implementáció
- A fordítás optimalizálása - kapcsolók

GLOBAL MEMORY: Coalesced Memory access

Alapprobléma

A globális memória látens ideje 400-600 órajel (1 összeadás = 4 órajel)

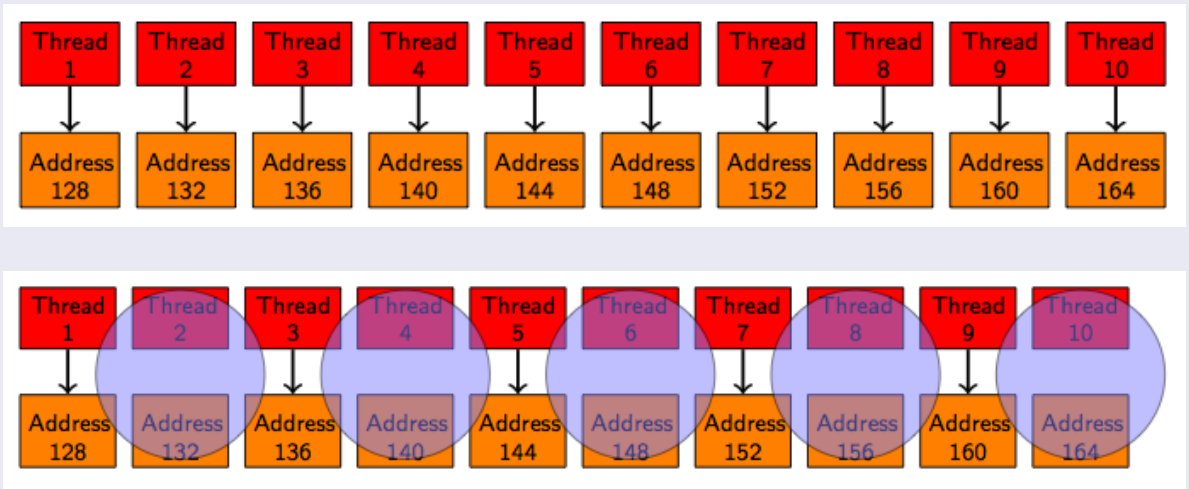
Fizikai futtatási modell: **warp** - a szálak fizikailag 32-es csokrokban futnak

A szálak hozzáférése a globális memóriához **rendezett**, amennyiben:

- A memóriatartomány összefüggő és **rendezett**:
  - **128 byte**: minden szál egy *float* v. egy *int* adatot olvas
  - **256 byte**: minden szál egy *float2* v. egy *int2* adatot olvas
  - **512 byte**: minden szál egy *float4* v. egy *int4* adatot olvas
  - a *float3* NEM rendezett!!!
- A *warp* olvasási báziscíme (*WBA*) *16 x sizeof(type)* többszöröse
- Egy csokron belül a *k*. szál éppen a (WBA+k). elemhez fér hozzá
- A memóriaműveletben nem muszáj minden szálnak részt venni
- A fenti szabályok írásra és olvasásra egyaránt vonatkoznak

GLOBAL MEMORY: Coalesced Memory access

Rendezett memória hozzáférés:



GLOBAL MEMORY: Coalesced Memory access

NEM rendezett memória hozzáférés:

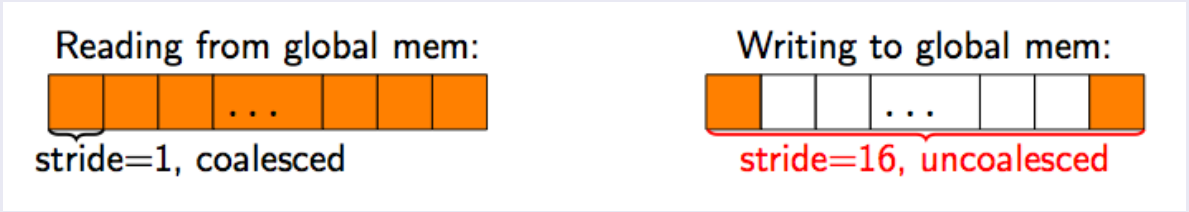
SHARED MEMORY

Feladat: Négyzetes $(n * n)$ mátrix transzponálása

Naiv GPU mag

```
__global__
void transpose_matrix_naive(float *in,float *out,int n){

int i=blockDim.x*blockIdx.x+threadIdx.x;
int j=blockDim.y*blockIdx.y+threadIdx.y;

if ((i<n)&&(j<n))
  out[i*n+j]=in[j*n+i];

}
```

Memóriahozzáférés



SHARED MEMORY

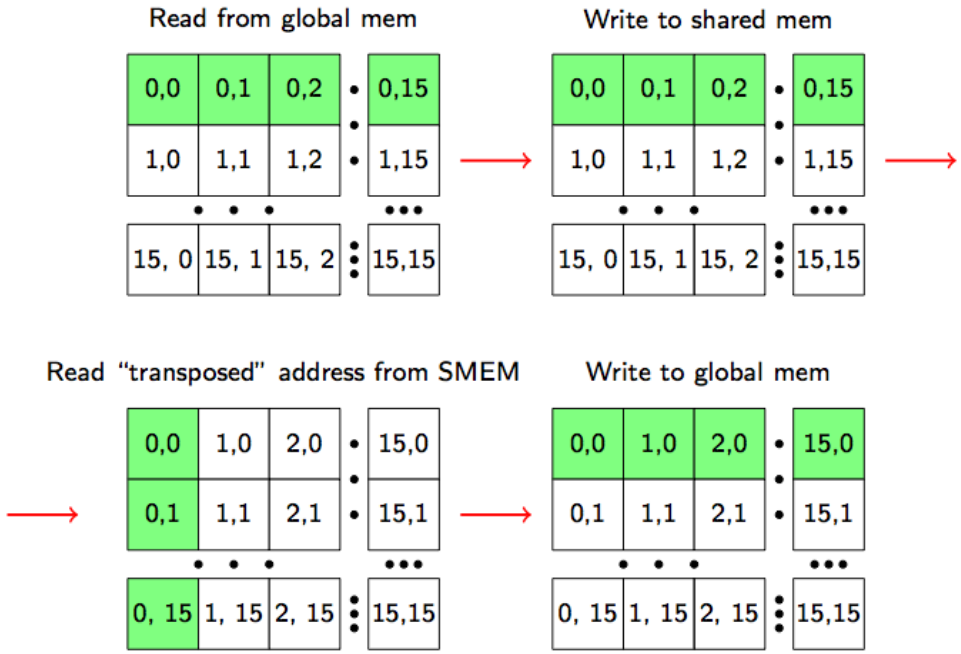
Megoldás a shared memory használatán keresztül

Naiv modell

- rendezett olvasás (gyors)
- NEM rendezett írás (lassú)

GPU modell

- rendezett olvasás:
  global → shared (gyors)
- transzponálás:
  shared → shared (gyors)
- rendezett írás:
  shared → global (gyors)



SHARED MEMORY

Gyors GPU mag

```
__global__
void transpose(float *in,float *out,int n) {
__shared__float block[BLOCK_DIM*BLOCK_DIM];

int xBlock=blockDim.x*blockIdx.x;
int yBlock=blockDim.y*blockIdx.y;
int xIndex=xBlock+threadIdx.x;
int yIndex=yBlock+threadIdx.y;

int index_out, index_transpose;

if ((xIndex<n)&&(yIndex<n)) {
  int index_in=n*yIndex+xIndex;
  int index_block=threadIdx.y*BLOCK_DIM+threadIdx.x;

  block[index_block]=in[index_in];

  index_transpose=threadIdx.x*BLOCK_DIM+threadIdx.y;
  index_out=n*(xBlock+threadIdx.y)+(yBlock+threadIdx.x);
}

__synchthreads();

if ((xIndex<n)&&(yIndex<height)) {
  out[index_out]=block[index_transpose];
}
}
```

Naiv GPU mag

```
__global__
void transpose_matrix_naive(float *in,float *out,int n){

int i=blockDim.x*blockIdx.x+threadIdx.x;
int j=blockDim.y*blockIdx.y+threadIdx.y;

if ((i<n)&&(j<n))
  out[i*n+j]=in[j*n+i];

}
```

Megérte?

Grid Size	Coalesced	Non-coalesced	Speedup
128 × 128	0.011 ms	0.022 ms	2.0×
512 × 512	0.07 ms	0.33 ms	4.5×
1024 × 1024	0.30 ms	1.92 ms	6.4×
1024 × 2048	0.79 ms	6.6 ms	8.4×

SHARED MEMORY

__syncthreads()

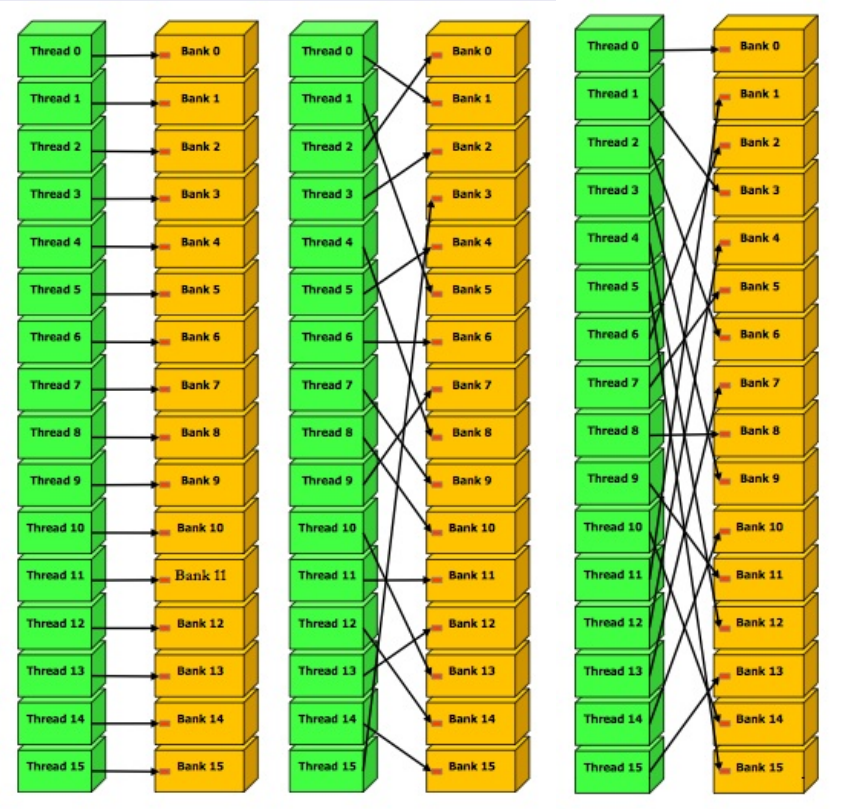
- Az egy blokkban futó szálak ennél az utasításnál "bevárják" egymást
- shared memory használat esetén a szálak tudnak "egymásnak" dolgozni:
  - transzponálás: egy szál $s(i, j)$-t írja, de $s(j, i)$-t olvassa!
  - molekuladinamika: egy szál az i. atom adatait tölti be ($i \in [0 \ldots THREADS - 1]$) de mindenkiét használja - összes párkölcsönhatás!

A shared memory hozzáférés modellje: bank conflict

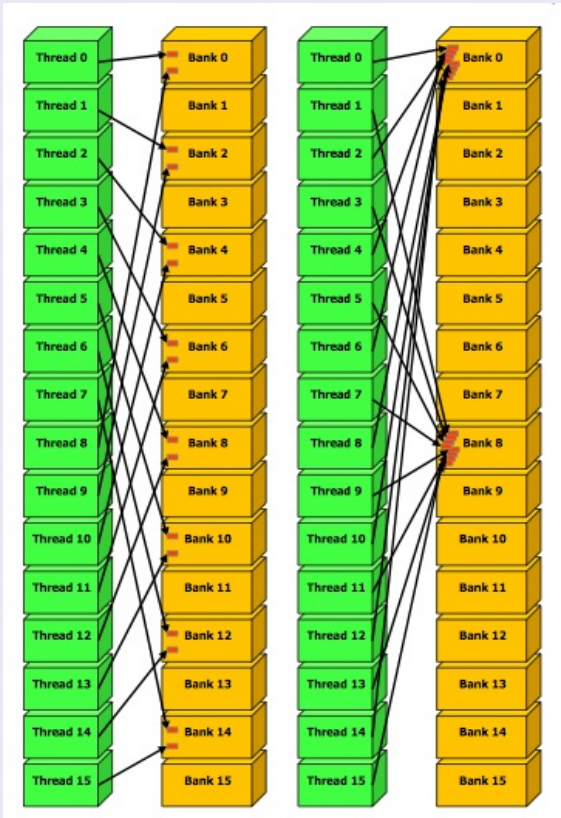
- Regiszter sávszélesség (32 bit / 2 órajel) biztosítása: shared memory = 16 x **bank**
- ideális hozzáférés: 1 szál $\Leftrightarrow$ 1 bank
- n-utas **bank conflict**: n szál $\rightarrow$ 1 bank ($\Rightarrow$ soros: n x 1 hozzáférés)
  warp = 32 szál $\Rightarrow$ $\rightarrow$ fordító $\rightarrow$ 2 x 16 felosztású hozzáférés
- A shared változó tárolási rendje: lineáris, 32 bites szavanként periodikusan $\Rightarrow$
  Egy lineáris *float* típusú tömb esetén biztosan NINCS bank conflict!

SHARED MEMORY - Bank conflict példák

No bank conflict



Bank conflict



Az IEEE-754 és a GPU

Az IEEE-754 szabvány

A számítástechnikai eszközök számítási szabványa

- digitális számábrázolási szabványok
- műveleti pontosság szabványai - kerekítés, vágás?

A GPU gyári utasításkészlete

- Alapfeladat: $+$ és $*$ gyors elvégzése $\Rightarrow$ kisebb átmeneti tár
- Utasításkészlet: jóval kisebb, mint a CPU-é, de van "mad": $a * b + c$
  $\rightarrow$ A fordító megteszi a lehetséges összevonásokat a kódból!

A GPU *float* műveleti pontossága - IEEE-754 kivételek

- $+$ és $*$: IEEE-754 pontos
- *mad*, valamint bonyolultabb NATIV műveletek ($1/x$, y/x, $\sqrt{x}$, $\exp(x)$, ...) pontatlanabbak (vágás)
- szoftveres megoldások a CPU pontosságának elérésére (de lassabbak...)!

Példa: elemenkénti $A * B + C$

"naiv" GPU mag

```
__global__
void mad(float *A,float *B,float *C,float *D,int n){

int whoami=blockDim.x*blockIdx.x+threadIdx.x;

if (whoami<n*n)
  D[whoami]=A[whoami]*B[whoami]+C[whoami];

}
```

IEEE-754 helyes GPU mag

```
__global__
void mad(float *A,float *B,float *C,float *D,int n){

int whoami=blockDim.x*blockIdx.x+threadIdx.x;

if (whoami<n*n)
  D[whoami]=__fmaf_rn(A[whoami],B[whoami],C[whoami]);

}
```

Gépi kód

$+$ és $*$ összevonása *mad* műveletté
$\Rightarrow$ $+$ és $*$ külön-külön IEEE-754 pontos, de a *mad* nem!

Gépi kód

Nincs összevonás fordításnál
__fmaf_xx() : IEEE-754 pontos mad
$\Rightarrow$ Lassabb, de pontosabb

A GPU programozás "kereskedelem": pontosság + gyorsaság $\approx$ const.

Műveleti idők

Nativ műveletek

- 4 clock cycles:
  - Floating point: add, multiply, fused multiply-add
  - Integer add, bitwise operations, compare, min, max
- 16 clock cycles:
  - reciprocal, reciprocal square root, `__log(x)`, 32-bit integer multiplication
- 32 clock cycles:
  - `__sin(x)`, `__cos(x)` and `__exp(x)`
- 36 clock cycles:
  - Floating point division (24-bit version in 20 cycles)
- Particularly costly:
  - Integer division, modulo
  - Remedy: Replace with shifting whenever possible
- Double precision (when available) will perform at half the speed

Tanács

Kerüljük az **osztást**, a maradékképzést, és a magasabb szintű függvényeket!

A feladat

- Adott N db részecske 3D-ben: $\{x_i, y_i, z_i\}$, $i = 1 \ldots N$.
- Készítsünk hisztogramot (h) a páronkénti távolságukból ($BINS, DR$)!
- Fizika: $g[(i + 1/2)DR] = \frac{2}{N} h[i]/dV_i$ (Radial Distribution Function)

CPU kód

```
void rdf(void){

int i,j;

for (i=0; i<N-1; i++)
  for (j=i+1; j<N; j++)
        if ((int)(dist(i,j)/DR)<BINS) h[idx]++;

}
```

CPU kód

```
float dist(int i,int j){

float dx=x[i]-x[j];
float dy=y[i]-y[j];
float dz=z[i]-z[j];

return(sqrtf(dx*dx+dy*dy+dz*dz));

}
```

Implementációs probléma

A változó hosszúságú belső ciklus nem jól párhuzamosítható!

Alternetív implementáció

- A részecskék koordinátái: $x[i], y[i], z[i]$ lineáris tömbök, $i = 0 \ldots N-1$
- $dist(i,j)$, ahol $j = (i+d)\%N$, ciklus $i = 0 \ldots N-1$, valamint $d = 1 \ldots N/2-1$-re

CPU kód I.

```
void rdf(void){

int i,j;

for (i=0; i<N-1; i++)
  for (j=i+1; j<N; j++)
        if ((int)(dist(i,j)/DR)<BINS) h[idx]++;

}
```

CPU kód II.

```
void rdf(void){

int d,i;

for (d=1; d<N/2; d++)
  for (i=0; i<N; i++)
   if ((int)(dist(i,(i+d)%N)/DR)<BINS) h[idx]++;

}
```

naiv GPU mag

```
__global__
void gpu_test_kernel(float *x,float *y,float *z,int
*h,int d){

int whoami=blockIdx.x*blockDim.x+threadIdx.x;
int idx;

if (whoami<N){
  idx=(whoami+d)%N;
  float3 p1,p2;
  p1.x=x[whoami]; p1.y=y[whoami]; p1.z=z[whoami];
  p2.x=x[idx];    p2.y=y[idx]; p2.z=z[idx];
  idx=(int)(__fdiv_rn(dist(p1,p2),DR));
  if (idx<BINS)
   atomicAdd(h+idx,1);
}

}
```

GPU kód

```
__device__
float dist(float3 p1,float3 p2){

float dx=p1.x-p2.x;
float dy=p1.y-p2.y;
float dz=p1.z-p2.z;

float dx2=__fmul_rn(dx,dx);
float dy2=__fmul_rn(dy,dy);
float dz2=__fmul_rn(dz,dz);

float tmp=__fadd_rn(dx2,dy2);
float d2=__fadd_rn(tmp,dz2);

float d=__fsqrt_rn(d2);

return(d);

}
```

Tulajdonságok:

- A GPU távolságszámoló függvénye és az osztás a magban CPU pontos (IEEE-754)
- Egy új függvény: *__atomicAdd(int *,int)* (sm_12)

Optimalizálás

A naiv GPU mag "hibái"

- A beolvasás, bár rendezett, 3 tömbre fut
- Központi probléma: a memóriahozzáférés a kiemeneten ($h[idx]++$)
  ADATFÜGGŐ
  A memória írás BIZTOSAN NEM rendezett, sőt,*atomic* függvényre van szükség.

Megoldás

- A rendezett beolvasás *float4* segítségével gyorsítható
  (*float3*-mal nem, mert az memóriaművelet szempontból NEM rendezett!)
- Definiáljunk al-hisztogramokat a blokkokon a shared memory-ba!
  $\rightarrow$ Az al-hisztogramokat a számítás végén rendezetten egyesítjük

Optimalizált RDF

(koránt sem naiv) GPU mag

```
__global__
void gpu_test_kernel(float4 *p,int *h,int d){

int whoami=blockIdx.x*blockDim.x+threadIdx.x;
int idx;

__shared__int sh[BINS];

for (int i=0; i<D; i++)
  sh[i*THREADS+threadIdx.x]=0;
__syncthreads();

if (whoami<N){
  idx=(whoami+d)%N;
  float4 p1=p[whoami];
  float4 p2=p[idx];
  idx=(int)(__fdiv_rn(dist(p1,p2),DR));
  if (idx<BINS)
   atomicAdd(sh+idx,1);
}
__syncthreads();

for (int i=0; i<D; i++){
  idx=i*THREADS+threadIdx.x;
  atomicAdd(h+idx,sh[idx]);
}

}
```

GPU kód

```
__device__
float dist(float4 p1,float4 p2){

float dx=p1.x-p2.x;
float dy=p1.y-p2.y;
float dz=p1.z-p2.z;

float dx2=__fmul_rn(dx,dx);
float dy2=__fmul_rn(dy,dy);
float dz2=__fmul_rn(dz,dz);

float tmp=__fadd_rn(dx2,dy2);
float d2=__fadd_rn(tmp,dz2);

float d=__fsqrt_rn(d2);

return(d);

}
```

- gyors shared műveletek
- bank conflict (random pattern)
- rendezett global műveletek

Teszteredmények

$N = 44028, BINS/THREADS = 1, THREADS = 512$

- 1 CPU mag: 58.23 s
- GPU naiv (*float4*, no shared): 16.3 s (3.57x)
- GPU optim. 1 (*float4*, shared): 1.9 s (30.6x)
- GPU optim. 2 (teljes probléma kernel): $< 1s$ (> 60x)

Üzenet: $\tau_{CPU}^{single}/\tau_{GPU}^{ideal} > 60$ (Fermi + CUDA 4.0: ≈ 500!)

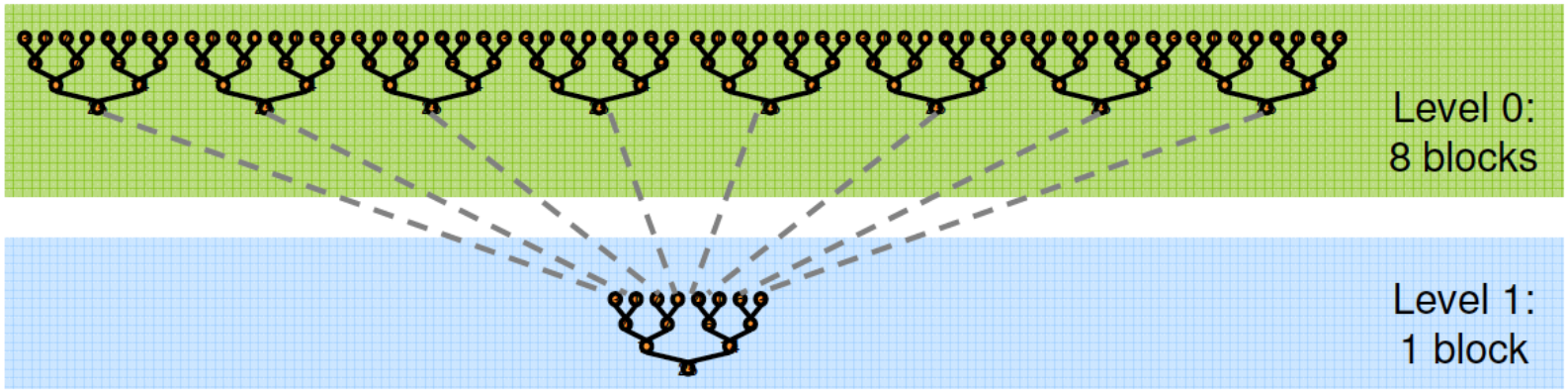
A kimeneti oldalon rosszul kondícionált probléma ellenére úgy tűnik, megérte!

Megjegyzések

- Az IEEE-754 pontosság kikapcsolása 5.0%; (% $\rightarrow$*if*) áttérés 2.5%
- shared memory $\rightarrow \tau \approx 1/THREADS \Rightarrow$ A rendezetlen írás a szűk keresztmetszet
- Teljes probléma kernel, *__atomicAdd()* $\rightarrow$ bitwise, $\cdots \rightarrow \ldots$

A feladat

Adjuk össze egy lineáris tömb elemeit párhuzamos GPU kóddal!



A GPU kernel feladata

- Adatlefedés: N block $\times$ T thread $\rightarrow$ T db elem részösszege blokkonként
- Bemenet $\rightarrow$ kimenet: $N \times T$ hosszú lineáris tömb $\rightarrow$ N hosszú lineáris tömb
- Rekurzív (iteratív) kernel hívás a részösszegek tömbjén, míg előall a teljes összeg

A feladat alapszintű megoldása

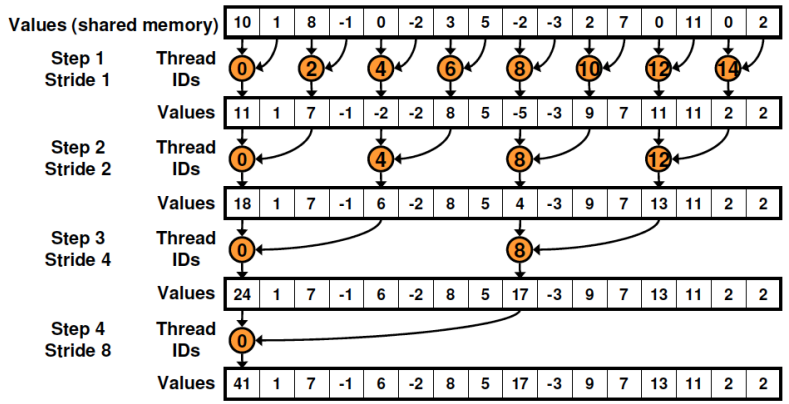
$T = 2^p$ elemszámú tömb elemeinek összeadása párhuzamos redukcióval

Naiv kernel

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Az eljárás elemei

- A tömb beolvasása a shared memory-ba
- Az elemek összeadása ciklikus redukcióval
- Az eredmény kiírása a globális memóriába



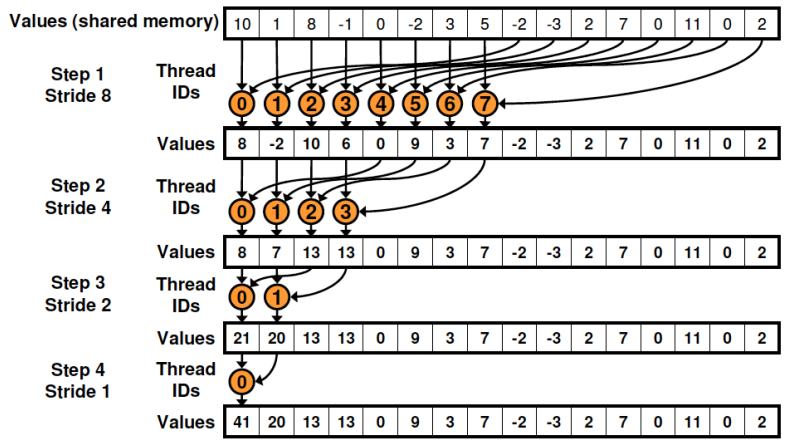
Optimalizálás

"Az állatorvosi ló"

1. divergent warps + %: nem egymás utáni thread-ek futnak + % sokba kerül
2. shared memory bank conflict: több thread egyszerre akar ugyanahhoz a területhez férni
3. Idle threads: a thread-ek fele eleve nem csinál semmit a beolvasáson kívül
4. loop unrolling #1: az utolsó warp-ban sok a felesleges művelet
5. loop unrolling #2: ha tudnánk a szükséges iterációk számát, meg lehetne szüntetni a ciklust
6. complexity problem: túl kevés aritmetikai müvelet vs. túl sok memóriaművelet / thread

Sok kicsi sokra megy!

1. Első 2^n dolgozik - 2.33×
2. stride-olt összeadás - 2.01×
3. 1. összeadás beolvasáskor - 1.78×
4. utolsó warp kifejtés - 1.8×
5. teljes ciklus kifejtés - 1.41×
6. algoritmus kaszkád - 1.41×

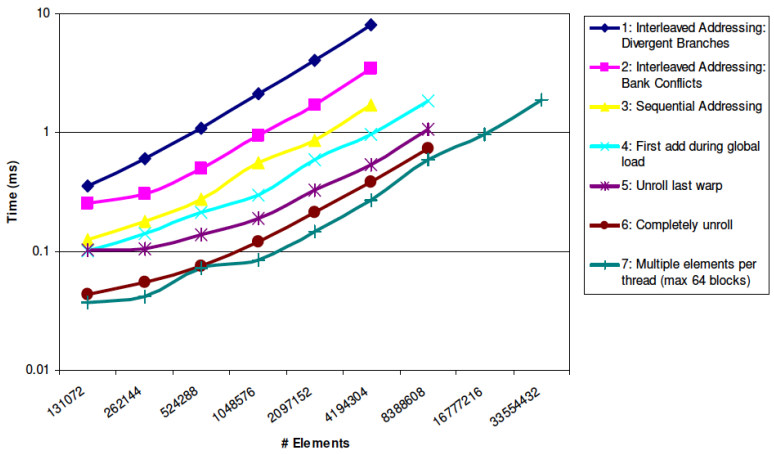


Eredmény

Performance

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!



Az optimális kernel

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;

  while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
  __syncthreads();

  if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
  if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
  if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

  if (tid < 32) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
  }

  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

35

Üzenet: 30× gyorsabb, de...

"Nem kezdőnek való vidék"

Összefoglalás

A GPU programozás optimalizálása

- A feladat: adatpárhuzamosítás (**grid**) + művelet párhuzamosítás (**kernel**)
- "Premature optimization is the root of all evil."
  $\Rightarrow$ Programozási modell: MAKE IT WORK. MAKE IT RIGHT. MAKE IT FAST.
- Az optimalizálás fő területei:
  - A feladat ideális implementációja az architekturához!
  - memóriakezelés (global és shared)- nagy gyorsulások
  - Egyéb: $\approx 2\times$ faktorok, de sok - nem kezdőknek való!
- Pontosság: A GPU gyors 3D ábrázoló, nem tudományos szuperszámítógép!
- MEGÉRI? - $\tau_{\text{feladatmagoldás}} = \tau_{\text{programfejlesztés}} + \tau_{\text{programfuttatás}}$
- GPU vs. CPU verseny: gyors fejlődés, érdemes várni a kész megoldásokra!