

Základní operace

Paralelní redukce

Paralelní redukce na architekturách se sdílenou pamětí

Paralelní redukce na architekturách s distribuovanou pamětí

Paralelní redukce na GPU v CUDA

Prefix sum

Segmentovaný prefix-sum

Co je paralelní redukce?

Definition

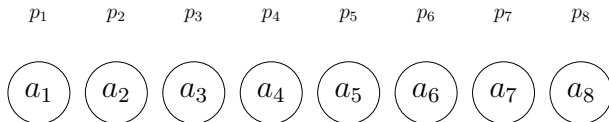
Mějme pole prvků a_1, \dots, a_n určitého typu a asociativní operaci \oplus . **Paralelní redukce** je paralelní aplikace asociativní operace \oplus na všechny prvky vstupního pole tak, že výsledkem je jeden prvek a stejného typu, pro který platí

$$a = a_1 \oplus a_2 \dots a_n.$$

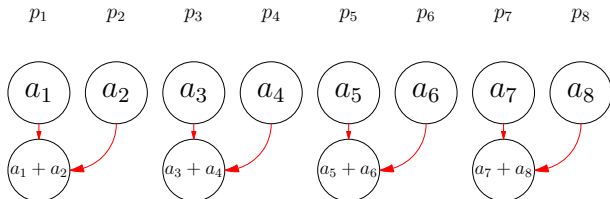
Příklad: Pro jednoduchost budeme uvažovat operaci sčítání. Pak jde o paralelní provedení tohoto kódu:

```
a = a[ 1 ];  
for( int i = 2; i <= n; i ++ )  
    a += a[ i ];
```

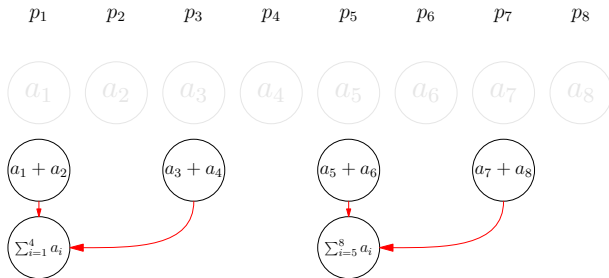
Paralelní provedení



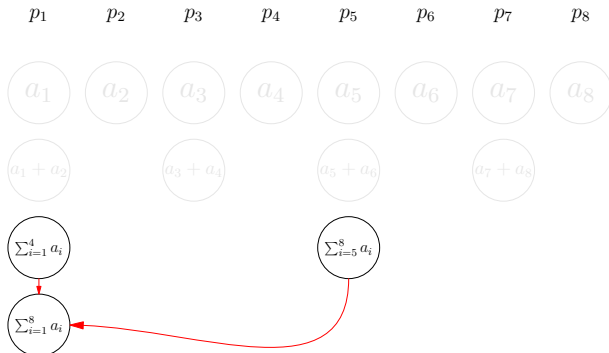
Paralelní provedení



Paralelní provedení



Paralelní provedení



Analyza paralelní provedení

Za předpokladu $n = p$, snadno vidíme, že platí:

- ▶ $T_S(n) = \theta(n)$
- ▶ $T_P(n) = \theta(\log n)$
- ▶ $S(n) = \theta\left(\frac{n}{\log n}\right) = O(n) = O(p)$
- ▶ $E(n) = \theta\left(\frac{1}{\log n}\right)$
 - ▶ pro velká n efektivita klesá k nule
- ▶ $C(n) = \theta(n \log n) = \omega(T_S(n))$
 - ▶ algoritmus není nákladově optimální

Provedeme úpravu algoritmu tak, aby byl nákladově optimální.

Nákladově optimální redukce

p_1



p_2



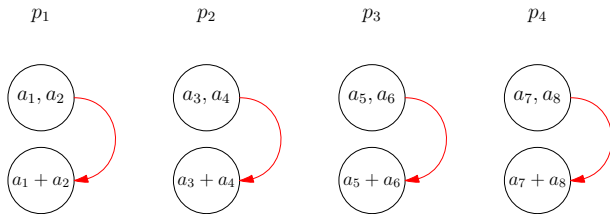
p_3



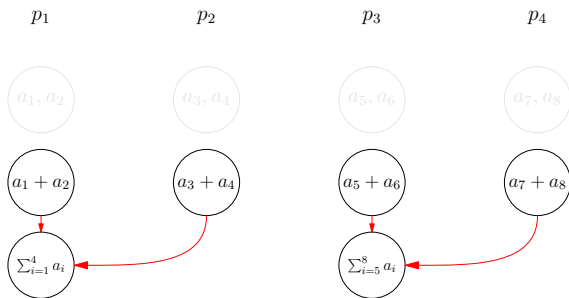
p_4



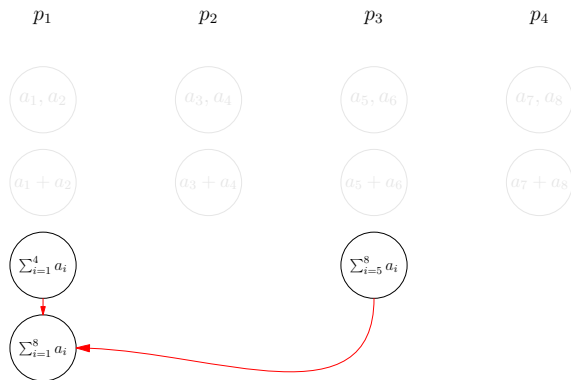
Nákladově optimální redukce



Nákladově optimální redukce



Nákladově optimální redukce



Nákladově optimální redukce

- ▶ nyní máme méně procesorů než prvků, tj. $p < n$.
- ▶ nejprve se provede sekvenční redukce $\frac{n}{p}$ prvků na každém procesoru $\rightarrow \theta\left(\frac{n}{p}\right)$
- ▶ a pak paralelní redukce na p procesorech $\rightarrow \theta(\log p)$
- ▶ celkem tedy $T_P(n, p) = \theta\left(\frac{n}{p} + \log p\right)$
- ▶ $S(n, p) = \frac{T_S(n)}{T_P(n, p)} = \theta\left(\frac{\frac{n}{p}}{\frac{n}{p} + \log p}\right) = \theta\left(\frac{p}{1 + \frac{p \log p}{n}}\right)$
- ▶ $E(n, p) = \frac{S(n, p)}{p} = \theta\left(\frac{1}{1 + \frac{p \log p}{n}}\right)$

Nákladově optimální redukce

Pak platí

- ▶ $C(n, p) = pT_P(n, p) = \theta(n + p \log p)$
- ▶ protože je $n = \Omega(p \log p)$, máme

$$C(n, p) = \theta(n) = \theta(T_S(n)).$$

Algoritmus je tedy nákladově optimální.

- ▶ toho jsme dosáhli na úkor počtu procesorů
- ▶ některé algoritmy nedokáží optimálně využít velký počet procesorů

Paralelní redukce na architekturách se sdílenou pamětí

- ▶ OpenMP má podporu pro redukci s základními operacemi
 - `reduction(sum: +)`
- ▶ pokud chceme dělat redukci s jinou operací (`min`, `max`) nebo jiným, než základním typem (matice), je nutné ji provést explicitně
- ▶ tyto architektury mají většinou maximálně několik desítek procesorů
- ▶ redukci lze často provádět sekvenčně
- ▶ mezivýsledky se ukládají do nesdílených proměnných
 - ▶ pozor na cache coherence a false sharing - tyto proměnné by neměly být alokované v jednom bloku
- ▶ nakonec se zapíše do sdíleného pole, na kterém se provede redukce třeba i sekvenčně nultým procesem

Paralelní redukce na architektúrah s distribuovanou pamětí

- ▶ standard MPI má velmi dobrou podporu pro redukci
- ▶ podporuje více operací a to i pro odvozené typy a struktury

Paralelní redukce na GPU v CUDA

Mějme pole o N prvcích. Paralelní redukce na GPU probíhá následujícím způsobem:

- ▶ redukci bude provádět $N/blkSize$ bloků
 - ▶ pro jednoduchost předpokládáme, že $blkSize$ dělí N
- ▶ výsledkem bude pole o $N/blkSize$ prvcích
- ▶ na toto pole opět provedeme stejným způsobem paralelní redukci, až dojdeme k poli o velikosti 1
 - ▶ ve skutečnosti se může vyplatit provést poslední redukci o pár prvcích na CPU
- ▶ úlohu jsme tak převedli na redukci dat v rámci jednoho bloku

Paralelní redukce na GPU v CUDA

```
1  __global__ void reductionKernel1( int* dOutput, int* dInput )
2  {
3      extern __shared__ volatile int sdata[];
4
5      // Nacteme data do sdilene pameti
6      unsigned int tid = threadIdx.x;
7      unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
8      sdata[ tid ] = dInput[ gid ];
9      __syncthreads();
10
11     // Provedeme paralelni redukci
12     for( unsigned int s=1; s<blockDim.x; s*=2 )
13     {
14         if( (tid % ( 2*s ) ) == 0 )
15             sdata[ tid ] += sdata[ tid + s ];
16         __syncthreads();
17     }
18
19     // Vysledek zapiseme zpet do globalni pameti zarizeni
20     if( tid == 0 )
21         dOutput[ blockIdx.x ] = sdata[ 0 ];
22 }
```

Paralelní redukce na GPU v CUDA

- ▶ `dInput` je vstupní pole pro redukci
- ▶ `dOutput` je výstupní pole pro redukci
- ▶ řádek č. 3 provádí dynamickou alokaci paměti ve sdílené paměti multiprocessoru do pole `sdata`
- ▶ proměnná `tid` udává číslo vlákna v rámci bloku
- ▶ proměnná `gid` udává číslo vlákna v rámci gridu a tím i prvek globálního pole `uInput`, který bude vlákno načítat
- ▶ na řádce 8 se načítají prvky do sdílené paměti
 - ▶ z globální paměti čteme sekvenčně, tedy pomocí sloučených přístupů (coalesced accesses)
- ▶ následně se na řádce 9 synchronizují, nelze redukovat, dokud nejsou načtena všechna data

Paralelní redukce na GPU v CUDA

- ▶ for cyklus na řádku 12 udává pomocí proměnné s , s jakým krokem redukuje
- ▶ začínáme s krokem jedna a krok se pokaždé zdvojnásobuje
- ▶ řádek 14 říká, že redukci provádějí vlákna, jejichž ID je dělitelné dvojnásobkem momentálního redukčního kroku
- ▶ vlastní redukce se provádí na řádku 15
- ▶ než pokračujeme dalším kolem redukce, synchronizujeme všechna vlákna na řádku 16
- ▶ nakonec, když je vše zredukované, nulté vlákno zapíše výsledek na správné místo do výstupního pole

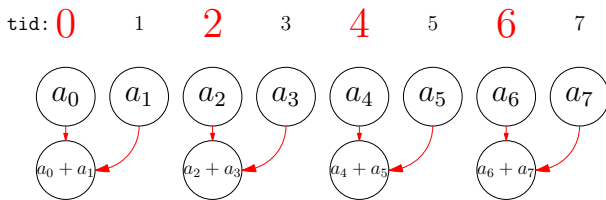
Paralelní redukce na GPU v CUDA

Tento postup ovšem není ideální. Dává následující výsledky (výkon měříme v GB dat zredukovaných za 1 sec.)

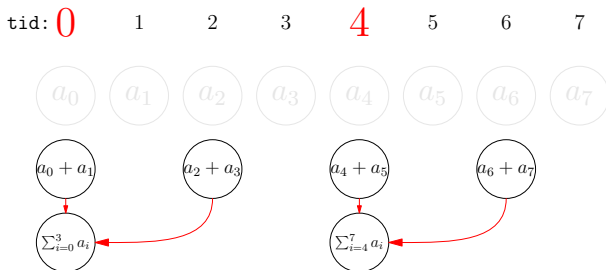
- ▶ GeForce GTX 8800 - 2.16 GB/sec
- ▶ GeForce GTX 460 - 4.26 GB/sec

Kde je problém?

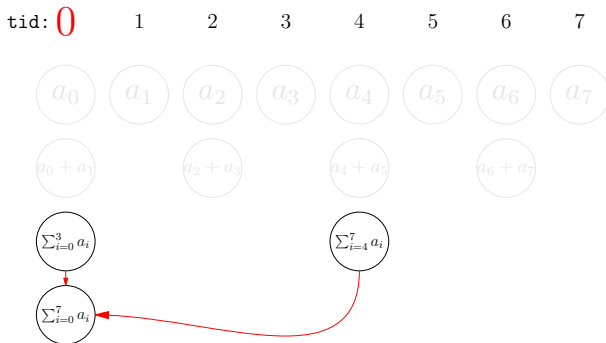
Paralelní redukce na GPU v CUDA



Paralelní redukce na GPU v CUDA



Paralelní redukce na GPU v CUDA



Paralelní redukce na GPU v CUDA

- ▶ používáme silně divergentní vlákna
 - ▶ pokaždé je více než jedna polovina warpu nečinná
- ▶ používáme pomalou funkci modulo

Paralelní redukce na GPU v CUDA

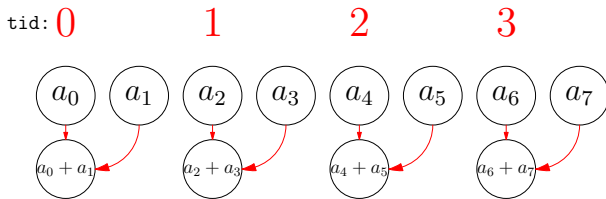
Nahradíme tento kód

```
1 // Provedeme paralelni redukci
2 for( unsigned int s=1; s < blockDim.x; s*=2 )
3 {
4     if( ( tid % ( 2 * s ) ) == 0 )
5         sdata[ tid ] += sdata[ tid + s ];
6     __syncthreads();
7 }
```

tímto

```
1 // Provedeme paralelni redukci
2 for( unsigned int s=1; s<blockDim.x; s*=2 )
3 {
4     unsigned int inds = 2*tid*s;
5     if( inds < blockDim.x )
6         sdata[ inds ] += sdata[ inds + s ];
7 }
8 __syncthreads();
9 }
```

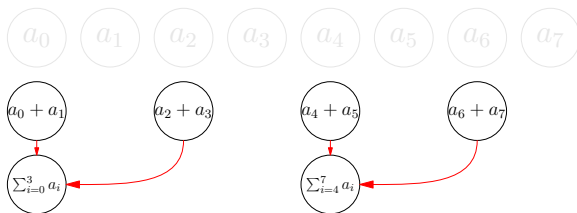
Paralelní redukce na GPU v CUDA



Paralelní redukce na GPU v CUDA

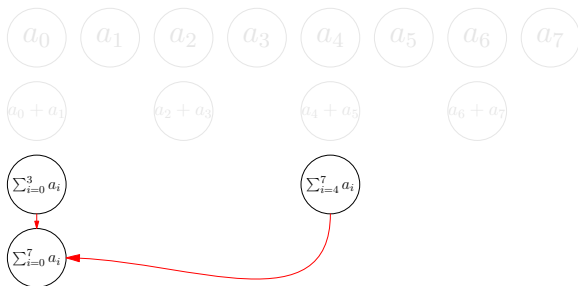
tid: **0**

1



Paralelní redukce na GPU v CUDA

tid: 0



Paralelní redukce na GPU v CUDA

Nyní nemapujeme vlákna podle indexu v původním poli, ale tak, že redukce vždy provádí prvních `blockDim.x/s` vláken.

- ▶ pokud redukuje jedním blokem 256 prvků, je v prvním kroku činných prvních 128 vláken a zbylých 128 ne
- ▶ první 4 warpy jsou plně vytížené, zbylé 4 warpy vůbec
- ▶ v žádném warpu ale nejsou divergentní vlákna

Paralelní redukce na GPU v CUDA

Nyní nemapujeme vlákna podle indexu v původním poli, ale tak, že redukce vždy provádí prvních `blockDim.x/s` vláken.

- ▶ pokud redukuje jedním blokem 256 prvků, je v prvním kroku činných prvních 128 vláken a zbylých 128 ne
- ▶ první 4 warpy jsou plně vytížené, zbylé 4 warpy vůbec
- ▶ v žádném warpu ale nejsou divergentní vlákna

Jak se změnil výkon?

- ▶ GeForce GTX 8800 - 4.28 GB/sec
- ▶ GeForce GTX 460 - 7.81 GB/sec

Výkon se zvýšil na dvojnásobek, což dobře koresponduje s tím, že jsme zabránili nečinnosti poloviny vláken ve warpu.

Kde je problém teď?

- ▶ v přístupech do sdílené paměti

Paralelní redukce na GPU v CUDA

- ▶ víme, že sdílená paměť se skládá z 32 paměťových bank
- ▶ pole wordů (4 bajty) se ukládá postupně do jednotlivých po sobě jdoucích bank
- ▶ pokud je $s = 32$ a redukuje typ `int`, pak všechna vlákna warpu čtou z jedné banky, ale každé z jiné adresy
- ▶ to je pochopitelně velmi pomalé

Paralelní redukce na GPU v CUDA

Nahradíme tento kód

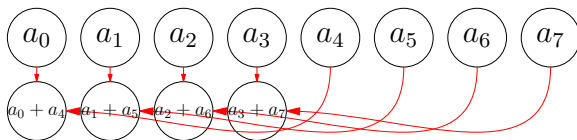
```
1 // Provedeme paralelni redukci
2 for( unsigned int s=1; s<blockDim.x; s*=2 )
3 {
4     unsigned int inds = 2*tid*s;
5     if( inds < blockDim.x )
6         sdata[ inds ] += sdata[ inds + s ];
7 }
8 __syncthreads();
9 }
```

tímto

```
1 // Provedeme paralelni redukci
2 for( unsigned int s=blockDim.x/2; s>0; s>>=1 )
3 {
4     if( tid < s )
5         sdata[ tid ] += sdata[ tid + s ];
6     __syncthreads();
7 }
```

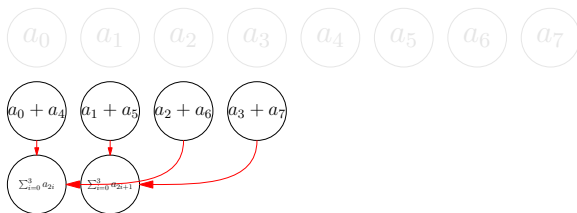

Paralelní redukce na GPU v CUDA

tid: 0 1 2 3



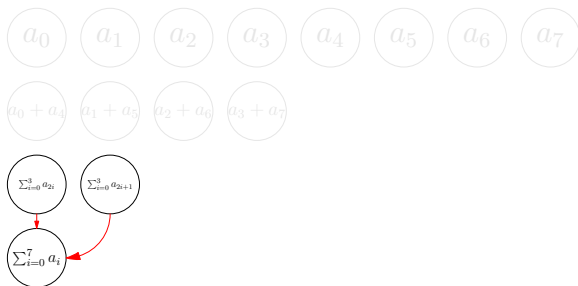
Paralelní redukce na GPU v CUDA

tid: 0 1



Paralelní redukce na GPU v CUDA

tid: 0



Paralelní redukce na GPU v CUDA

Redukce nyní probíhá tak, že slučujeme jeden prvek z první poloviny redukovaného pole a jeden prvek z druhé poloviny. Tím pádem vlákna přistupují i do sdílené paměti sekvenčně.

Paralelní redukce na GPU v CUDA

Redukce nyní probíhá tak, že slučujeme jeden prvek z první poloviny redukovaného pole a jeden prvek z druhé poloviny. Tím pádem vlákna přistupují i do sdílené paměti sekvenčně. Jak se změnil výkon?

- ▶ GeForce 8800 GTX - 9.73 GB/s
- ▶ GeForce 460 GTX - 11.71 GB/s

Co zlepšit dále?

- ▶ stále platí, že polovina vláken nedělá vůbec nic kromě načítání dat z globální paměti
- ▶ zmenšíme proto rozměr gridu na polovinu a necháme každé vlákno provést jednu redukci už při načítání dat z globální paměti

Paralelní redukce na GPU v CUDA

Nahradíme tento kód

```
1 // Nacteme data do sdílené paměti
2 unsigned int tid = threadIdx.x;
3 unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
4 sdata[ tid ] = dInp[ gid ];
5 __syncthreads();
```

tímto

```
1 unsigned int tid = threadIdx.x;
2 unsigned int gid = blockIdx.x*blockDim.x*2 + threadIdx.x;
3 sdata[ tid ] = dInp[ gid ] + dInp[ gid+blockDim.x ];
4 __syncthreads();
```

Paralelní redukce na GPU v CUDA

Poznámka: Pozor! Používáme poloviční grid. V kódu to není explicitně vidět. Kernel se ale musí volat s polovičním počtem bloků!!!

Jak se změnil výkon?

- ▶ GeForce 8800 GTX - 17.6 GB/s
- ▶ GeForce 460 GTX - 23.43 GB/s

Paralelní redukce na GPU v CUDA

V dalším kroku odstraníme for cyklus:

```
1   for( unsigned int s=blockDim.x/2; s>0; s>>=1 )
```

Jelikož maximální velikost bloku je 1024, proměnná *s* může nabývat jen hodnot

- ▶ 512, 256, 128, 64, 32, 16, 8, 4, 2 a 1.

Kernel nyní přepíšeme s pomocí šablon C++.

Paralelní redukce na GPU v CUDA

```
1  template <unsigned int blockSize>
2  __global__ void reductKern5( int* dOutput, int* dInput )
3  {
4      extern __shared__ volatile int sdata[];
5
6      // Nacteme data do sdilene pameti
7      unsigned int tid = threadIdx.x;
8      unsigned int gid = blockIdx.x*blockDim.x*2 + threadIdx.x;
9      sdata[ tid ] = dInput[ gid ] + dInput[ gid+blockDim.x ];
10     __syncthreads();
11
12     // Provedeme paralelni pyramidalni redukci
13     if (blockSize == 1024) {
14         if( tid<512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
15     if (blockSize >= 512) {
16         if( tid<256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
17     if (blockSize >= 256) {
18         if( tid<128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
19     if (blockSize >= 128) {
20         if( tid< 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __syncthreads(); }
21     if (tid < 32) {
22         if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
23         if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
24         if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
25         if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
26         if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
27         if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
28     }
29
30     // Vysledek zapiseme zpet do globalni pameti zarizeni
31     if (tid == 0) dOutput[blockIdx.x] = sdata[0];
32 }
```

Paralelní redukce na GPU v CUDA

- ▶ podle toho, jak je blok velký, se provede potřebný počet redukcí
- ▶ ve chvíli, kdy redukcí již provádí jen 32 nebo méně vláken, výpočet probíhá v rámci jednoho warpu
- ▶ vlákna v jednom warpu jsou synchronizována implicitně neboť jde čistě o SIMD zpracování, proto již nemusíme volat `__syncthreads()`

Jak se změnil výkon?

- ▶ GeForce 8800 GTX - 36.7 GB/s
- ▶ GeForce 460 GTX - 46.8 GB/s

Co lze zlepšit dále?

Paralelní redukce na GPU v CUDA

- ▶ nyní pomocí p vláken redukuje $2p$ prvků
- ▶ ukázali jsme si ale, že v tom případě není výpočet nákladově optimální
- ▶ to znamená, že používáme příliš velký počet procesorů, které ale nejsou efektivně využity
- ▶ v případě GPU to znamená, že multiprocesory nejsou stále plně využity
- ▶ když budeme určitý úsek redukovaného pole zpracovávat menším počtem multiprocesorů, bude jejich využití efektivnější, a zbylé multiprocesory se uvolní pro zpracovávání zbytku redukovaných dat
- ▶ pro docílení nákladové optimality je potřeba volit $p = \frac{n}{\log n}$
 - ▶ pro pole o velikosti 2048 prvků dostáváme $p = \frac{2048}{11} = 341$
 - ▶ výsledek zaokrouhlíme na 256

Výsledný kernel vypadá takto:

Paralelní redukce na GPU v CUDA

```
1  template <unsigned int blockSize>
2  __global__ void reductKern6(int* dOutput, int* dInput, uint size)
3  {
4      extern __shared__ volatile int sdata[];
5      // Nacteme data do sdilene pameti
6      unsigned int tid = threadIdx.x;
7      unsigned int gid = blockIdx.x*blockSize*2 + threadIdx.x;
8      unsigned int gridSize = blockSize*2*gridDim.x;
9      sdata[tid] = 0;
10     while (gid < size)
11     {
12         sdata[tid] += dInput[gid] + dInput[gid+blockSize];
13         gid += gridSize;
14     }
15     __syncthreads();
16
17     // Provedeme paralelni pyramidalni redukci
18     if (blockSize == 1024) {
19         if ( tid < 512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
20     if (blockSize >= 512) {
21         if ( tid < 256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
22     if (blockSize >= 256) {
23         if ( tid < 128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
24     if (blockSize >= 128) {
25         if ( tid < 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __syncthreads(); }
26     if (tid < 32) {
27         if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
28         if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
29         if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
30         if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
31         if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
32         if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
33     }
34
35     // Vysledek zapiseme zpet do globalni pameti zarizeni
36     if (tid == 0) dOutput[blockIdx.x] = sdata[0];
37 }
```

Paralelní redukce na GPU v CUDA

Jak se změnil výkon?

- ▶ GeForce 8800 GTX - 57.2 GB/s
- ▶ GeForce 460 GTX - ??? GB/s

Další optimalizace pro architekturu Kepler:

[http://devblogs.nvidia.com/parallelforall/
faster-parallel-reductions-kepler/](http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/)

Paralelní redukce na GPU v CUDA

Kernel	GTX 8800		GTX 460	
	výkon GB/s	urychlení	výkon GB/s	urychlení
Kernel 1 divergentní vlákna	2.16	–	4.3	–
Kernel 2 přístupy do sdílené paměti	4.28	1.98	7.8	1.8
Kernel 3 nečinná vlákna	9.73	2.3/ 4.5	11.71	1.5/ 2.7
Kernel 4 for cyklus	17.6	1.8/ 8.1	23.43	2/ 5.4
Kernel 5 není nákladově optimální	36.7	2.1/ 16.9	46.8	2/ 10.8
Kernel 6	57.2	1.6/ 26.5		

	diverg. vl.	L1 konflikty	aktivní warpy	spuštěné warpy
Kernel 1 divergentní vlákna	112,344	0	90,435,920	18,728
Kernel 2 přístupy do sdílené paměti	4,689	211,095	48,277,598	18,764
Kernel 3 nečinná vlákna	4,687	0	37,313,361	18,736
Kernel 4 for cyklus	2,343	0	19,501,609	9,376
Kernel 5 není nákladově optimální	294	0	5,948,100	1,180
Kernel 6	???	???	???	???

Prefix sum

Definition

Mějme pole prvků a_1, \dots, a_n určitého typu a asociativní operaci \oplus . **Inkluzivní prefix sum** je aplikace asociativní operace \oplus na všechny prvky vstupního pole tak, že výsledkem je pole s_1, \dots, s_n stejného typu, pro kterou platí

$$s_i = \bigoplus_{j=1}^i a_j.$$

Exkluzivní prefix sum je definován vztahy $\sigma_1 = 0$ a

$$\sigma_i = \bigoplus_{j=1}^{i-1} a_j,$$

pro $i > 0$.

Prefix sum

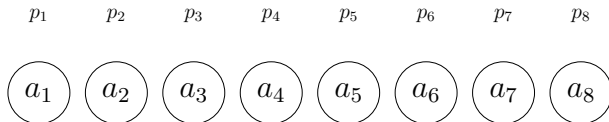
Poznámka: $\sigma_i = s_i - a_i$ pro $i = 1, \dots, n$.

Příklad: Pro jednoduchost budeme uvažovat operaci sčítání.

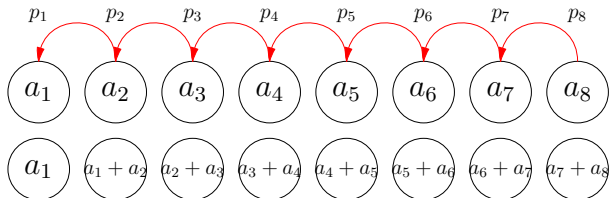
Pak jde o paralelní provedení tohoto kódu:

```
s[ 1 ] = a[ 1 ];  
for( int i = 2; i <= n; i ++ )  
    s[ i ] += a[ i ] + s[ i - 1 ];
```

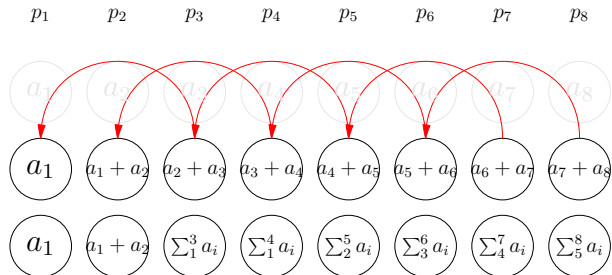

Paralelní prefix sum



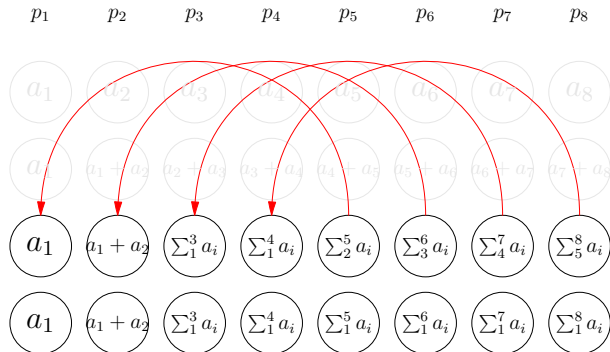
Paralelní prefix sum



Paralelní prefix sum



Paralelní prefix sum



Paralelní prefix sum

- ▶ $T_S(n) = \theta(n)$
- ▶ $T_P(n) = \theta(\log n)$
- ▶ $S(n) = \theta\left(\frac{n}{\log n}\right) = O(n) = O(p)$
- ▶ $E(n) = \theta\left(\frac{1}{\log n}\right)$

Paralelní prefix sum

- ▶ vidíme, že prefix sum má stejnou složitost jako redukce
- ▶ abysme získali nákladově optimální algoritmus, potřebovali bysme na jeden procesor mapovat celý blok čísel
- ▶ nyní je to o trochu složitější

Nákladově optimální paralelní prefix sum

- ▶ volíme $p < n$, pro jednoduchost tak, aby p dělilo n
- ▶ nechť každý procesor má blok A_k pro $k = 1, \dots, p$
- ▶ v něm se sekvenčně napočítá částečný prefix sum

$$s_i^* = \sum_{j \in A_i \wedge j \leq i} a_j.$$

- ▶ definujeme posloupnost S_k pro $k = 1, \dots, p$ tak, že $S_k = s_{i_k}$, kde i_k je index posledního prvku v bloku A_k .
- ▶ napočítáme exkluzivní prefix sum z posloupnosti $S_k \rightarrow \Sigma_k$
- ▶ každý procesor nakonec přičte Σ_k ke svým s_i
- ▶ potom je $T_P(n, p) = \theta \left(\frac{n}{p} + \log p \right)$ stejně jako u redukce

Paralelní prefix sum na GPU

Prefix sum v rámci jednoho bloku:

```
1  template< typename Real >
2  __global__ Real prefixSum( Real* values )
3  {
4      int i = threadIdx. x;
5      int n = blockDim. x;
6
7      for( int offset=1; offset<n; offset << 1 )
8      {
9          Real t;
10         if( i>=offset ) t=values[ i-offset ];
11         __syncthreads();
12
13         if( i>=offset) values[ i ] += t;
14         __syncthreads();
15     }
16 }
```


Paralelní prefix sum na GPU

- ▶ tato implementace opět není příliš efektivní, budeme optimalizovat
- ▶ na GPU jsme omezeni hierarchickou strukturou paměti
- ▶ nejprve ukážeme prefix sum v rámci jednoho warpu ...
- ▶ ... dál v rámci bloku ...
- ▶ ... a nakonec v rámci gridu
- ▶ využijeme k tomu variantu pro nákladově optimální prefix sum

Paralelní prefix sum na GPU pro jeden warp

```
1  template< ScanKind Kind, typename Real >
2  __device__ Real prefixSumWarp( volatile Real* values, int idx=threadIdx. x)
3  {
4      int lane = idx & 31;
5      // index of thread in warp
6
7      if( lane >= 1 ) values[idx] = values[ idx - 1 ] + values[ idx ];
8      if( lane >= 2 ) values[idx] = values[ idx - 2 ] + values[ idx ];
9      if( lane >= 4 ) values[idx] = values[ idx - 4 ] + values[ idx ];
10     if( lane >= 8 ) values[idx] = values[ idx - 8 ] + values[ idx ];
11     if( lane >= 16 ) values[idx] = values[ idx - 16 ] + values[ idx ];
12
13     if( Kind == inclusive ) return values[ idx ];
14     else return ( lane>0 ) ? values[ idx-1 ] : 0;
15 }
```

Paralelní prefix sum na GPU pro jeden blok

- ▶ pro jednoduchost předpokládejme, že maximální velikost bloku je druhá mocnina velikosti warpu ($32^2 = 1024$)
- ▶ nejprve se provede prefix sum v rámci jednotlivých warpů bloku
- ▶ uložíme si poslední prvek z každého warpu
- ▶ tím dostaneme pole o velikosti max. jednoho warpu
- ▶ provedeme opět prefix sum (exkluzivní) ve warpu
- ▶ každý warp si pak přičte svoji hodnotu z výsledku z předchozího kroku ke všem svým hodnotám
- ▶ pracujeme zde v rámci jednoho bloku, tedy v jednom kernelu a se sdílenou pamětí

Paralelní prefix sum na GPU pro jeden blok

```
1  template< ScanKind Kind, class T >
2  __device__ Real scanBlock( volatile Real* values, int idx=threadIdx. x )
3  {
4      extern __shared__ volatile int sdata[];
5
6      const int lane = idx & 31;
7      const int warpid = idx >> 5;
8
9      // Step 1: Intra-warp scan in each warp
10     Real val = scanWarp< Kind >( values, idx );
11     __syncthreads();
12
13     // Step 2: Collect per-warp partial results
14     if( lane == 31 ) sdata[ warpid ] = values[ idx ];
15     __syncthreads();
16
17     // Step 3: Use 1st warp to scan per-warp results
18     if( warpid == 0 ) scanWarp< exclusive >( sdata, idx );
19     __syncthreads();
20
21     // Step 4: Accumulate results from Steps 1 and 3
22     if( warpid > 0 ) val += sdata[ warpid ];
23     __syncthreads();
24
25     // Step 5: Write and return the final result
26     values[ idx ] = val;
27     __syncthreads();
28
29     return val;
30 }
```

Paralelní prefix sum na GPU pro jeden grid

- ▶ provedeme prefix sum pro každý blok
- ▶ poslední prvek každého bloku uložíme do pole `blockResults[]`
- ▶ na tomto poli se provede exkluzivní prefix sum
- ▶ pak si přičte i -tý blok ke všem svým prvkům hodnotu z i -tého prvku tohoto pole
- ▶ pokud výpočet probíhá ve více gridech, je potřeba toto zopakovat i v rámci gridů

Segmentovaný prefix-sum

- ▶ jde o napočítání několik prefix-sum najednou
- ▶ například

[1, 2, 3], [4, 5, 6, 7, 8]

dá výsledek

[1, 3, 6], [4, 9, 15, 22, 30]

- ▶ úlohu lze zakódovat jako jednu dlouhou řadu s restartovacími značkami
- ▶ sekvenční implementace pak může vypadat takto

Segmentovaný prefix-sum

```
1 void sequentialSegmentedPrefixSum( const int* a,  
2                                   const int* segments,  
3                                   const int n,  
4                                   int* segmentedPrefixSum )  
5 {  
6     segmentedPrefixSum[ 0 ] = a[ 0 ];  
7     for( int i = 1; i < n; i ++ )  
8         if( segments[ i ] == 0 )  
9             segmentedPrefixSum[ i ] =  
10                segmentedPrefixSum[ i - 1 ] + a[ i ];  
11     else  
12         segmentedPrefixSum[ i ] = a[ i ];  
13 }
```

Segmentovaný prefix-sum

- ▶ teoreticky lze segmentovaný prefix-sum implementovat stejně jako normální jen s jinak definovanou operací "sčítání"

$$(s_i, f_i) \oplus (s_j, f_j) = (f_j == 0 ? s_i + s_j : s_j, f_i | f_j),$$

kde $i < j$ a f_i je posloupnost jedniček na začátcích segmentů, jinak samé nuly