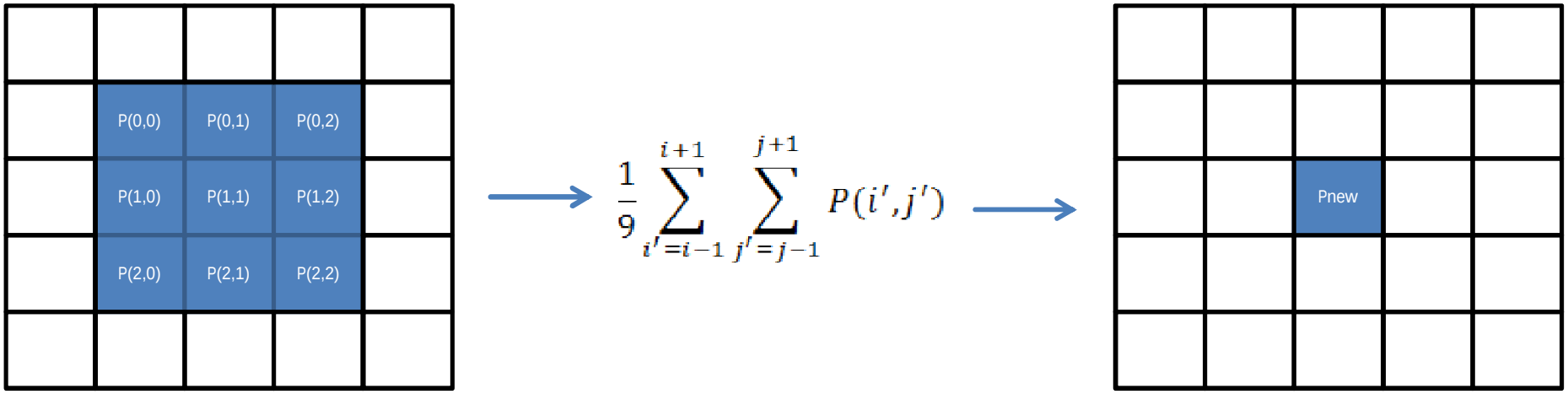
Optimizarea unei aplicații CUDA folosind memoria partajata (continuare)

Exemplu: Filtrarea unei imagini



Fiecare thread:

- Citește 9 valori din memoria globală
- Scrie o valoare în memoria globală

→ Fiecare valoare P(i,j) este citită de 9 thread-uri

Optimizarea unei aplicații CUDA folosind memoria partajata (continuare)

Exemplu: Filtrarea unei imagini

Pasul 1: Fiecare thread copiază valoarea P(i,j) ce-i corespunde din memoria globală în memoria partajată

Pasul 2: Se calculează media folosind valorile din memoria partajată

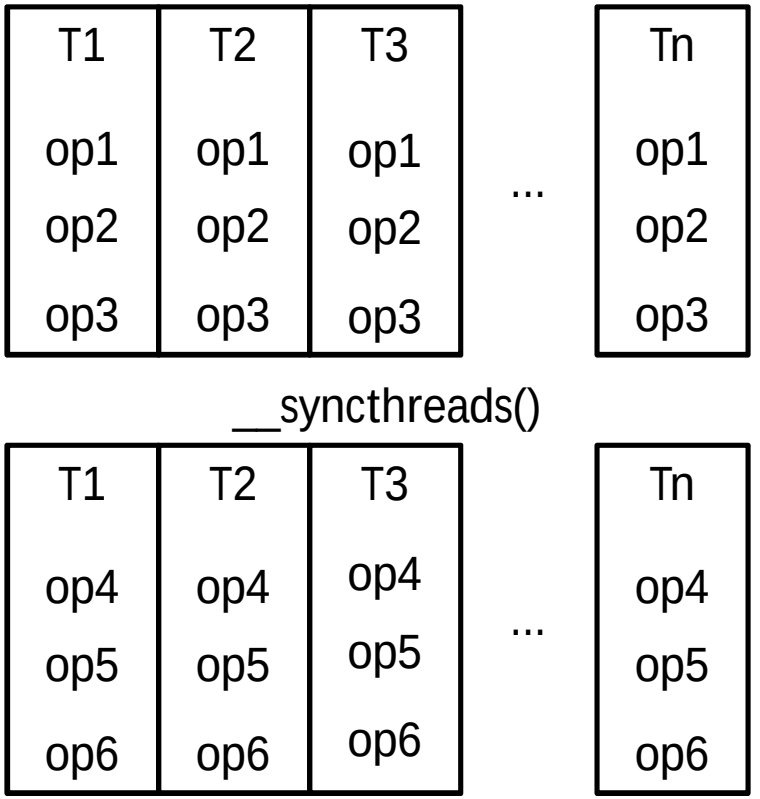
Pasul 3: Se scrie rezultatul în memoria globală

S-a redus numărul de citiri din memoria globală de la 9 la 1

Bariera de sincronizare
__syncthreads()

Sincronizează toate thread-urile dintr-un block

Orice thread din bloc va executa operațiile 4,5 și 6 după ce toate celelalte thread-uri au executat operatiile 1,2 și 3



Optimizarea unei aplicații CUDA folosind memoria partajata (continuare)

Exemplu: Filtrarea unei imagini

Pasul 1: Fiecare thread copiază valoarea P(i,j) ce-i corespunde din memoria globală în memoria partajată

Pasul 2: Barieră de sincronizare (*__syncthreads()*)

Pasul 3: Se calculează media folosind valorile din memoria partajată

Pasul 4: Se scrie rezultatul în memoria globală

S-a redus numărul de citiri din memoria globală de la 9 la 1

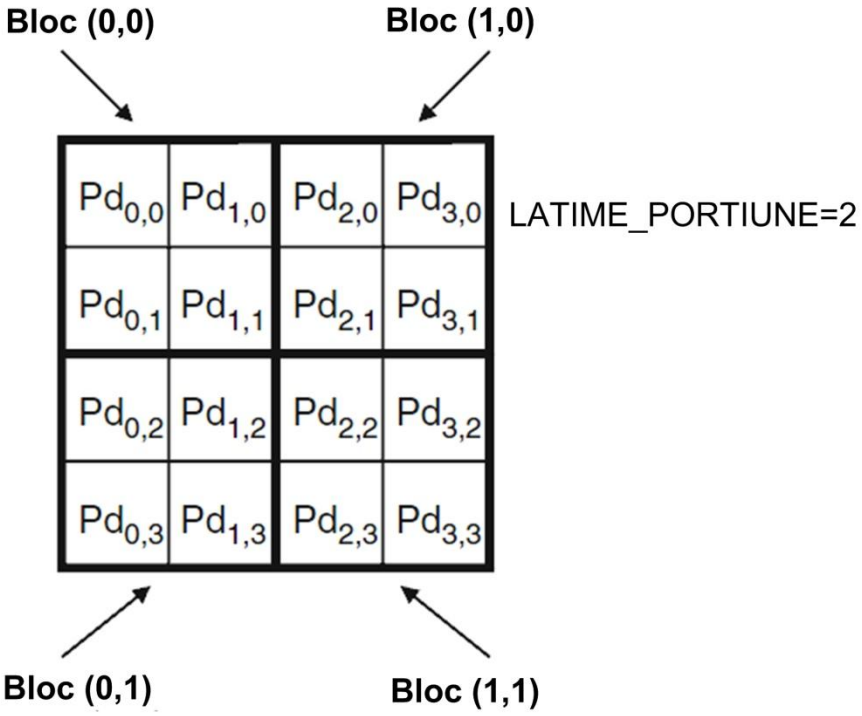
Optimizarea unei aplicații CUDA folosind memoria partajata (continuare)

Exemplu: Filtrarea unei imagini

Problema: Pixelii de la frontiera unui bloc

Copia imaginii în memoria partajată există doar la nivel de bloc → thread-urile de la “frontiera” unui bloc au nevoie de pixeli ce se află “în afară”

Soluții:

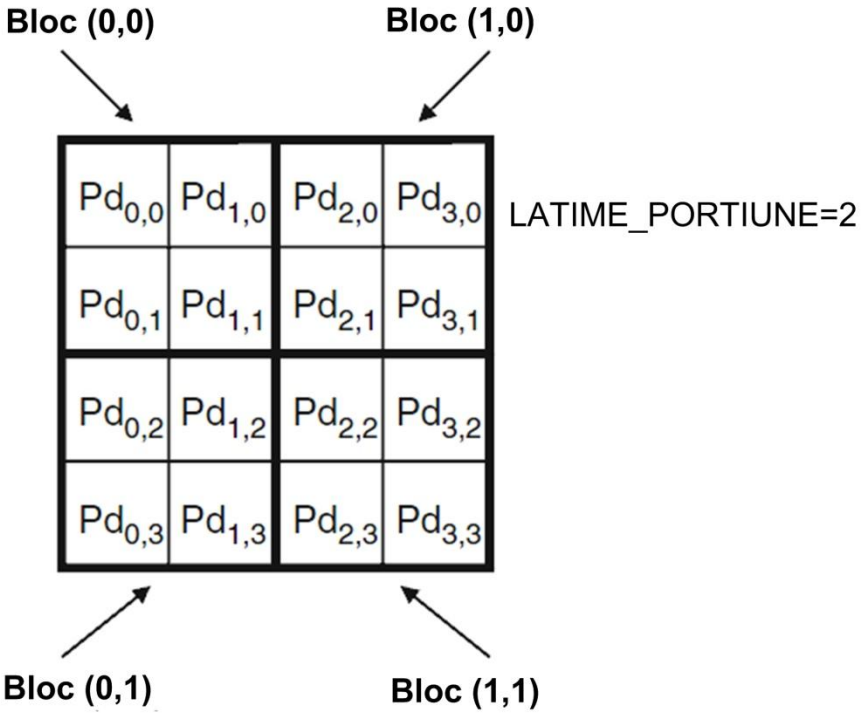


Optimizarea unei aplicații CUDA folosind memoria partajata (continuare)

Exemplu: Filtrarea unei imagini

Problema: Pixelii de la frontiera unui bloc

Copia imaginii în memoria partajată există doar la nivel de bloc → thread-urile de la “frontiera” unui bloc au nevoie de pixeli ce se află “în afară”



Soluții:

- Alocarea unui “strat” de pixeli în plus. Thread-urile de la frontieră vor copia pe lângă pixelul ce le corespunde și pixelul de “afară”
- Copierea din memoria globală a pixelilor ce nu există în memoria partajată

Accesarea memoriei globale (continuare)

- Pentru G80 viteza de accesare a mem. Globale este 86.4 GB/s iar numărul maxim de calcule în virgulă mobilă este 367 gflops (1.4 TB/s).
- Accesarea secvențială a memoriei globale.

Divergența codului

- Instrucțiuni *if-then-else*.
- If(threadIdx.x) vs. if(blockIdx.x)
- Structura repetitivă cu număr diferit de pași pentru fiecare thread din același warp.

Gradul de ocupare al unui GPU

- Grad de ocupare = numărul de thread-uri ce se execută simultan / numărul maxim ce poate fi executat simultan
- Factori ce limitează gradul de ocupare:
  - Numărul de thread-uri pe bloc
  - Numărul de regiștrii folosiți
  - Canitatea de memorie shared folosită

Numărul de thread-uri pe bloc

- Un gpu din arhitectura G80 poate executa maxim 8 blocuri simultan sau 1024 de thread-uri pe un multiprocesor.
- Dacă numărul de thread-uri pe bloc este prea mic gradul de ocupare al multiprocesorului va scădea.

Numărul de registrii

- Numărul de regiștrii pe multiprocesor este limitat.
- G80 are 16384 registrii pe multiprocesor.
- Dacă numărul maxim de regiștrii disponibili este depășit atunci se va reduce numărul de blocuri ce se execută simultan.

Cantitatea de shared memory

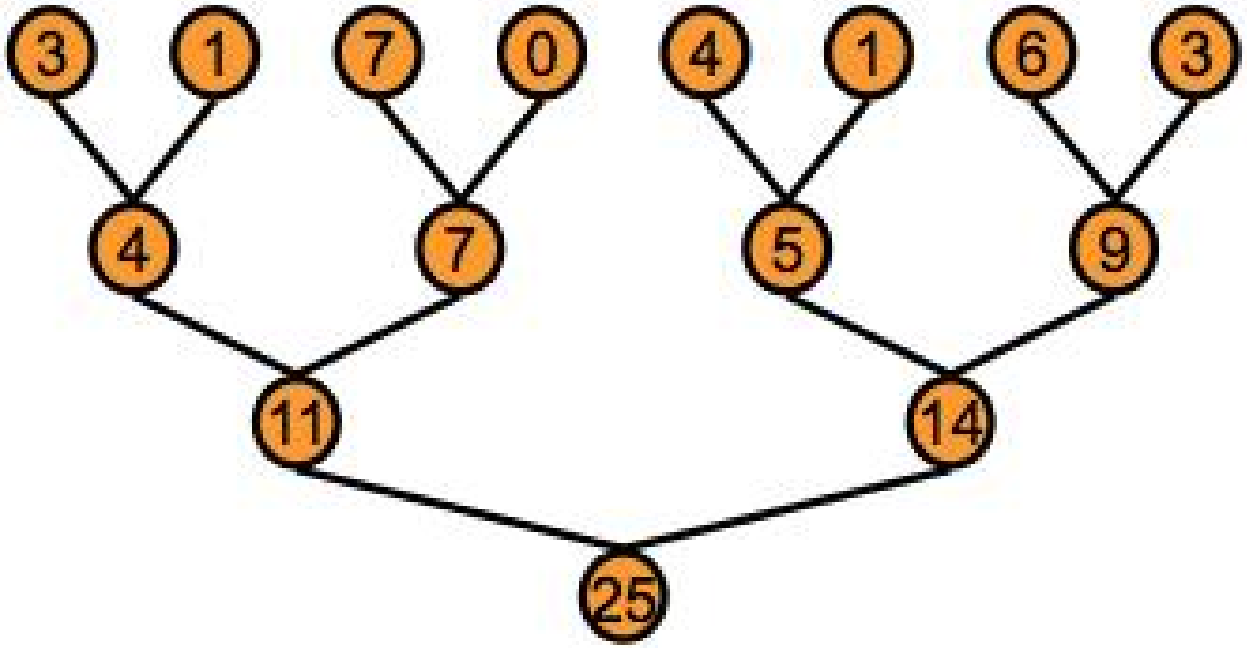
- Cantitatea de shared memory disponibilă pe multiprocesor este limitată.
- G80 are 16384 octeți de memorie shared.
- Dacă cantitatea de mem. Shared este depășită se va reduce numărul de thread-uri ce se exeută simultan.

Cantitatea de shared memory

- Cantitatea de shared memory disponibilă pe multiprocesor este limitată.
- G80 are 16384 octeți de memorie shared.
- Dacă cantitatea de mem. Shared este depășită se va reduce numărul de thread-uri ce se exeută simultan.
- Exemplu

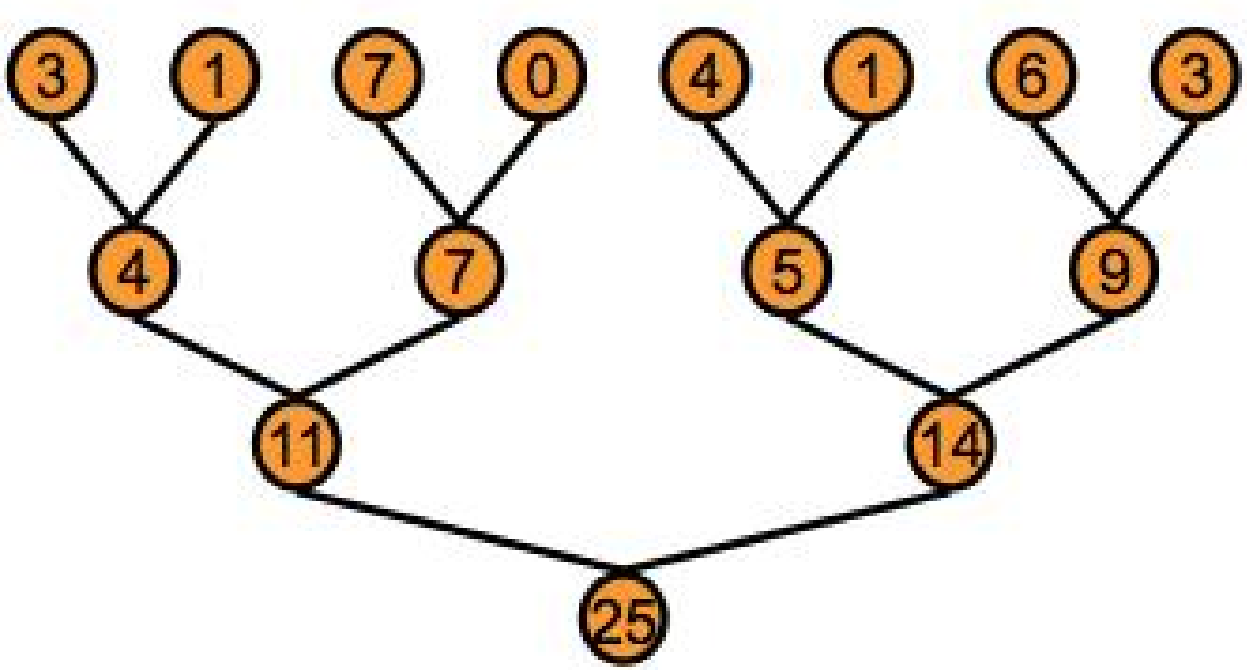
Reducere paralelă utilizând CUDA

Reducere paralelă utilizând CUDA



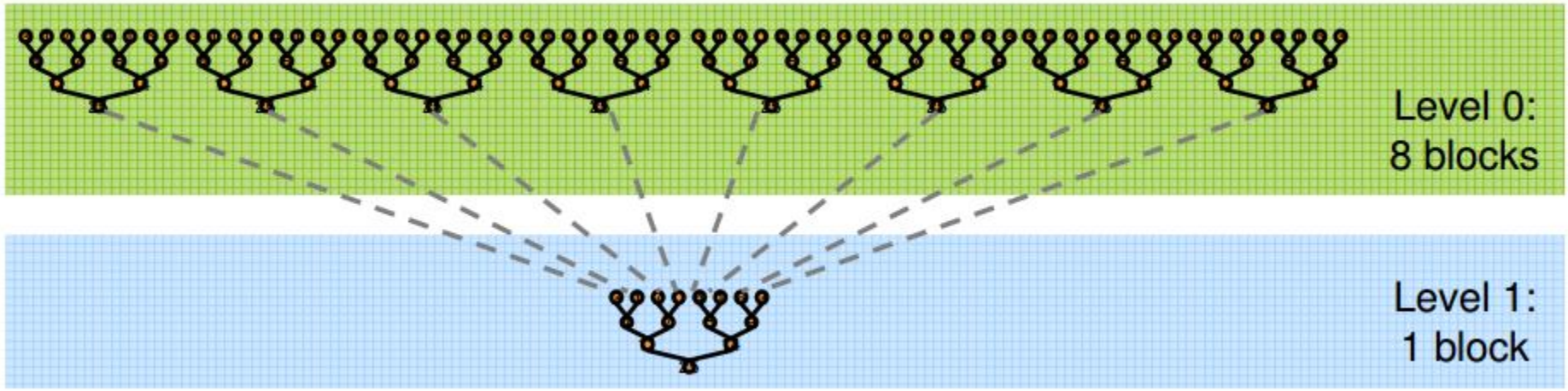
Reducere paralelă utilizând CUDA

- Problema – sincronizarea globală



Reducere paralelă utilizând CUDA

- Soluție – apelul kernelului de mai multe ori



Reducere paralelă

- Implementare: varianta 1

```
__global__ void sum( float *data, float *data_output, int size )
{
	__shared__ float sdata[BLOCK_SIZE];
	unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
	unsigned int tid = threadIdx.x;


	if( index >= size )
		return;


	sdata[tid] = data[index];
	__syncthreads();
	if( tid == 0 )
	{
		for( unsigned int i = 1; i < blockDim.x; i++ )
		{
				sdata[tid] += sdata[i];
		}
	  data[ blockIdx.x ] = sdata[0];

	}
}
```

Reducere paralelă

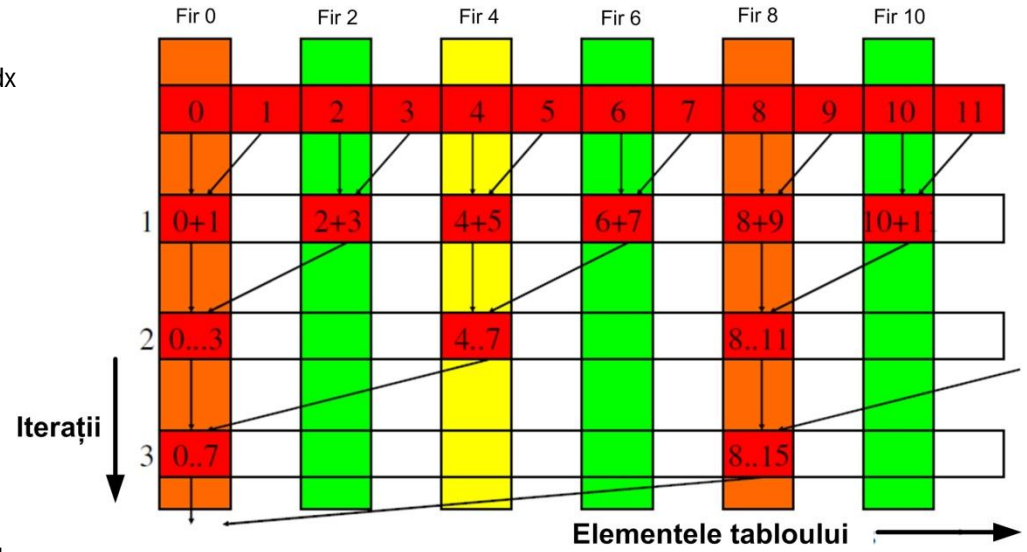
- Implementare: varianta 2

```
__global__ void sum( float *data, float *data_output, int size )
{
	__shared__ float sdata[128];
	unsigned int index = blockIdx.x * blockDim.x + threadIdx
	unsigned int tid = threadIdx.x;


	if( index >= size )
		return;


	sdata[tid] = data[index];
	__syncthreads();

	for(unsigned int s=1; s < blockDim.x; s *= 2)
	{
		if(tid % (2*s) == 0)
		{
			sdata[tid] += sdata[tid + s];
		}
		__syncthreads();
	}
	if(tid == 0)
		g_odata[blockIdx.x] = sdata[0];
}
```



Reducere paralelă

- Implementare: varianta 3

```
__global__ void sum( float *data, float *data_output, int size )
{
	__shared__ float sdata[128];
	unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
	unsigned int tid = threadIdx.x;

	if( index >= size )
		return;

	sdata[tid] = data[index];
	__syncthreads();
	for (unsigned int s = blockDim.x>>1; s > 0;s>>=1)
	{
		if (tid < s)
			sdata[tid] += sdata[tid+s];
	}

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