Lecture 18

Applications
Managing Locality

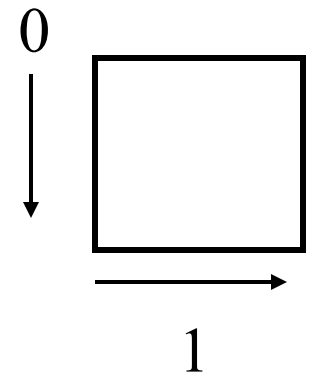
Today's lecture

- Short topic
  - Aliev-Panfilov on 64 cores of Triton
- GPUs
  - Under the hood of the device
  - Applications
  - Managing locality

Announcements

Performance is sensitive to processor geometry (Aliev-Panfilov on Triton.sdsc.edu)

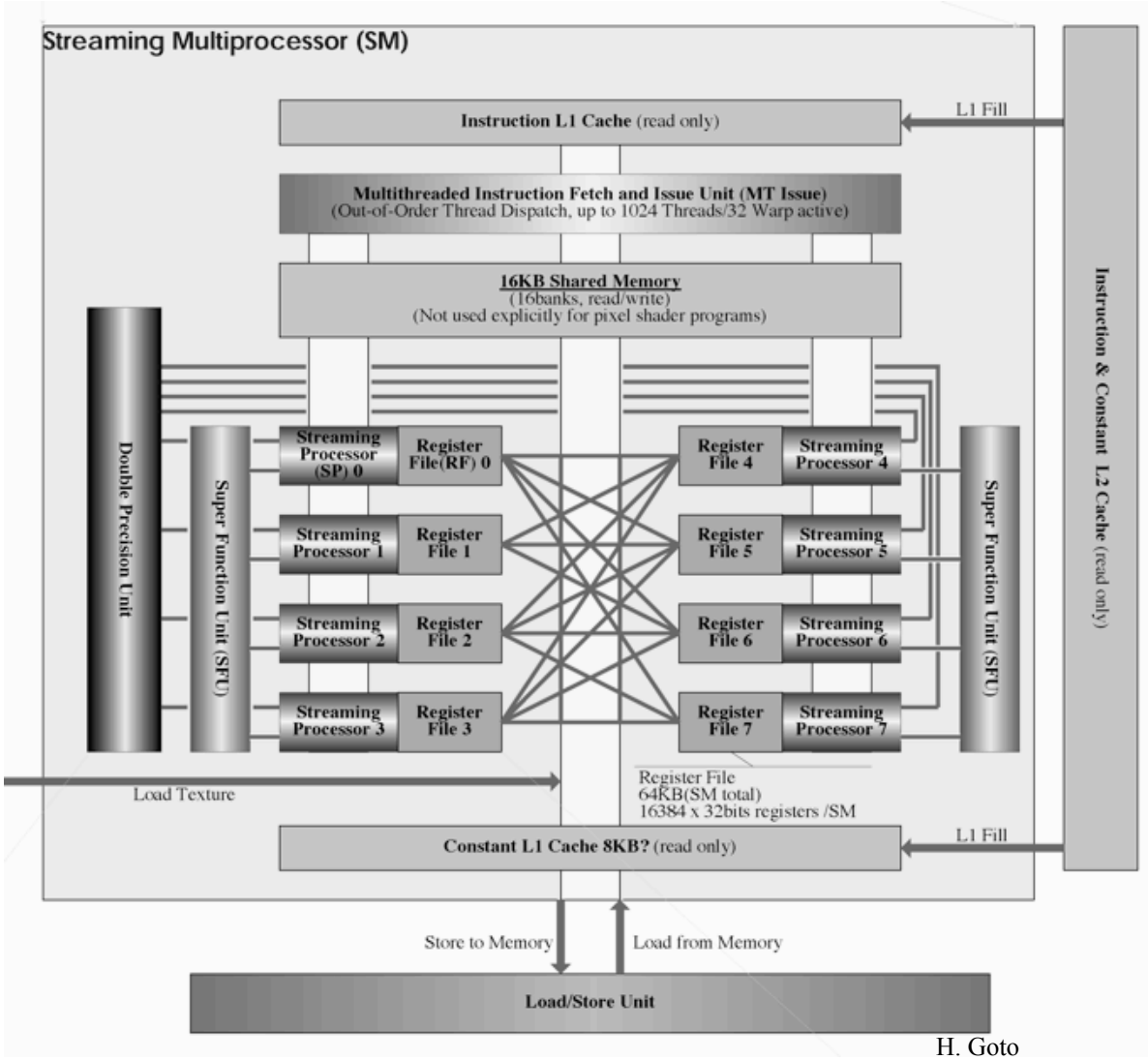
	Total			Without communication		
Geometry	GF/s	Time	Comm	GF/s	Time	
4 x 4	38.5	5.02	0.370	41.6	4.66	7.2% Communication
2 x 8	38.4	5.04	0.410	41.8	4.63	
8 x 2	38.1	5.07	0.376	41.2	4.70	
1 x 16	34.6	5.59	0.965	41.8	4.62	
16 x 1	33.0	5.87	1.00	39.8	4.87	
4x16	150.4	1.35	0.13	166.6	1.22	9.6% Communication
16x4	147.3	1.38	0.14	164.4	1.24	
8x8	146	1.4	0.17	165.9	1.23	
2x32	133.9	1.52	0.28	164.6	1.24	
1x64	130.9	1.56	0.32	164.2	1.24	
32x2	126.2	1.61	0.33	159.2	1.28	
64x1	119	1.71	0.33	147.8	1.38	



©2010 Scott B. Baden /CSE 260/ Fall '10

Under the hood of the device

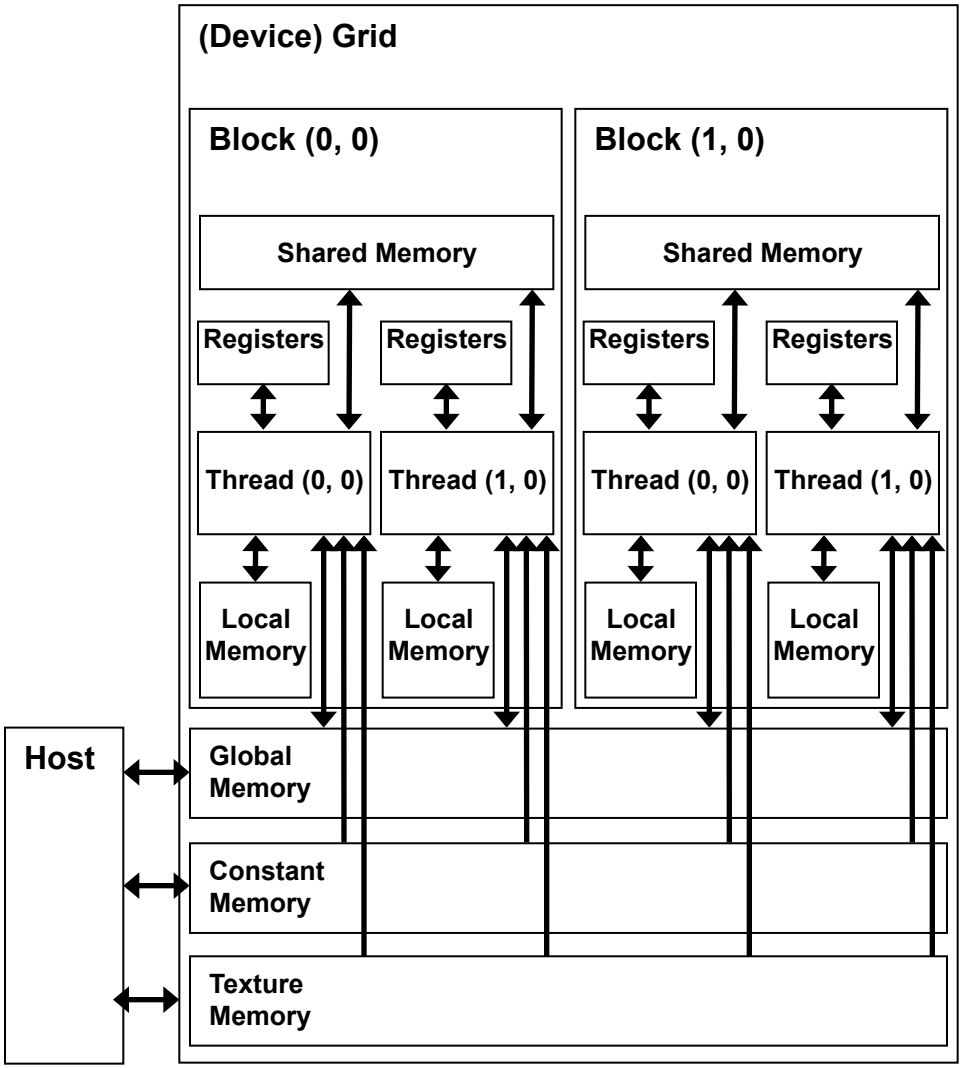
Streaming Multiprocessor



H. Goto

©2010 Scott B. Baden /CSE 260/ Fall '10

Memory Hierarchy



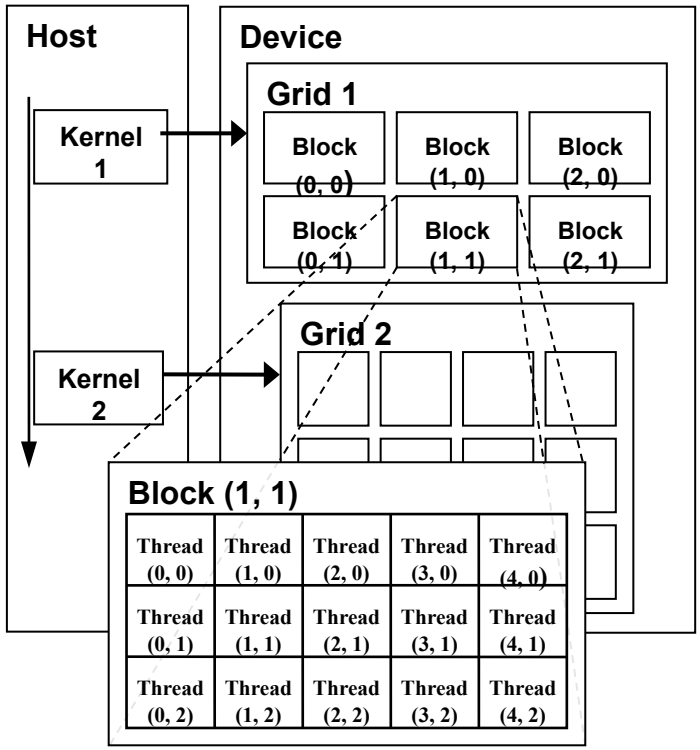
Name	Latency (cycles)	Cached
Global	DRAM – **100s**	No
Local	DRAM – 100s	No
Constant	1s – 10s – 100s	Yes
Texture	1s – 10s – 100s	Yes
Shared	1	--
Register	1	--

Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

©2010 Scott B. Baden /CSE 260/ Fall '10

Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
  - Grid ⊃ Block ⊃ Thread
- Thread Blocks
  - Subdivide a global index domain
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
- Threads
  - Assigned to SM in units of blocks
  - Compiler re-arranges loads to hide latencies
- Global synchronization via kernel invocation

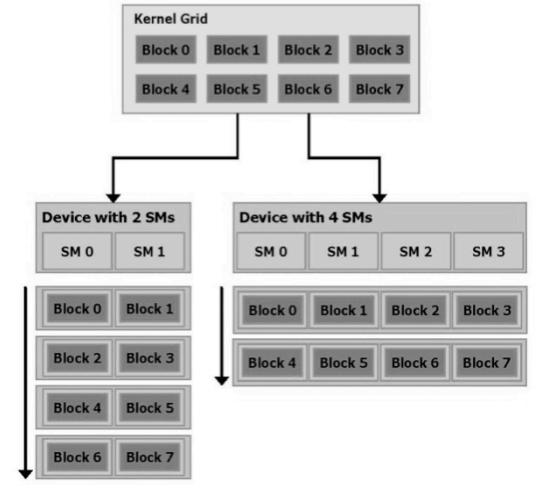


KernelA<<<2,3>,<3,5>>>()
Grid Block

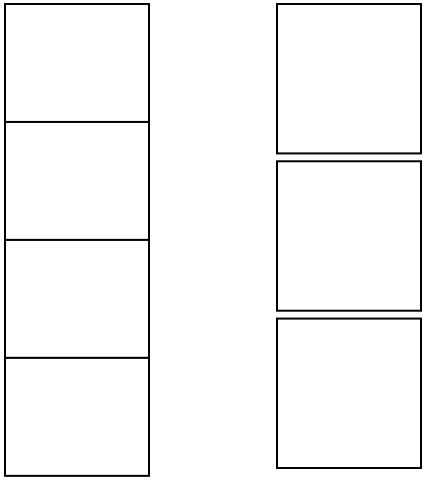
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Dynamic behavior – resource utilization

- Each vector core (SM):
  1024 thread slots and 8 block slots
- Hardware partitions slots into blocks at run time, accommodates different processing capacities
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block
- Bound to and only accessible from their thread until the block finishes execution

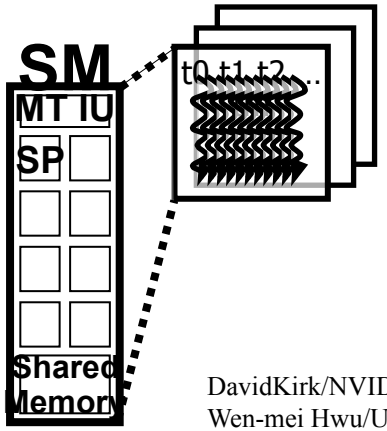


Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

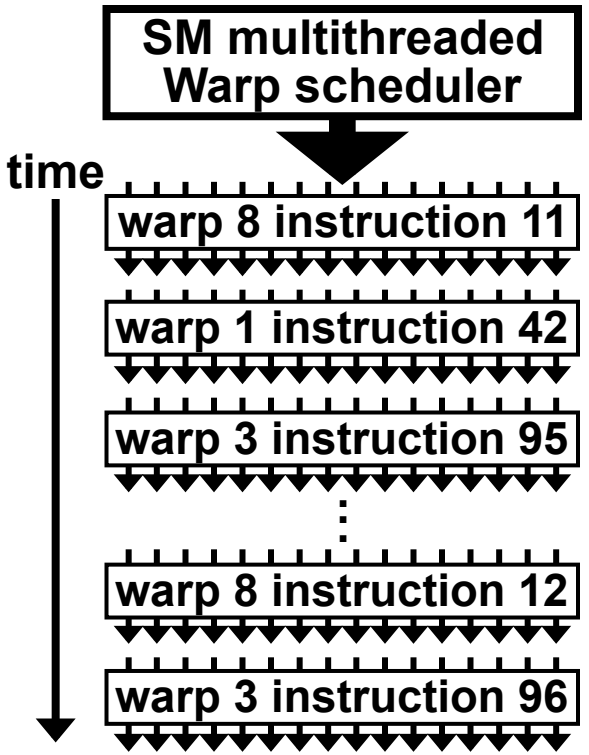


Warp scheduling

- Blocks are divided into *warps* of 32 (SIMD) threads which are ….
- Subidivided into schedulable units: 16 threads (→ 32 on Fermi)
- Zero scheduling overhead - in hardware
- Scheduler finds an eligible warp: all operands are ready
- Branches serialize execution in a warp
- To execute an instruction for all threads of a warp, the warp scheduler must issue the instruction over 4 clock cycles for an integer or single-precision floating-point arithmetic instruction [throughput = 8]

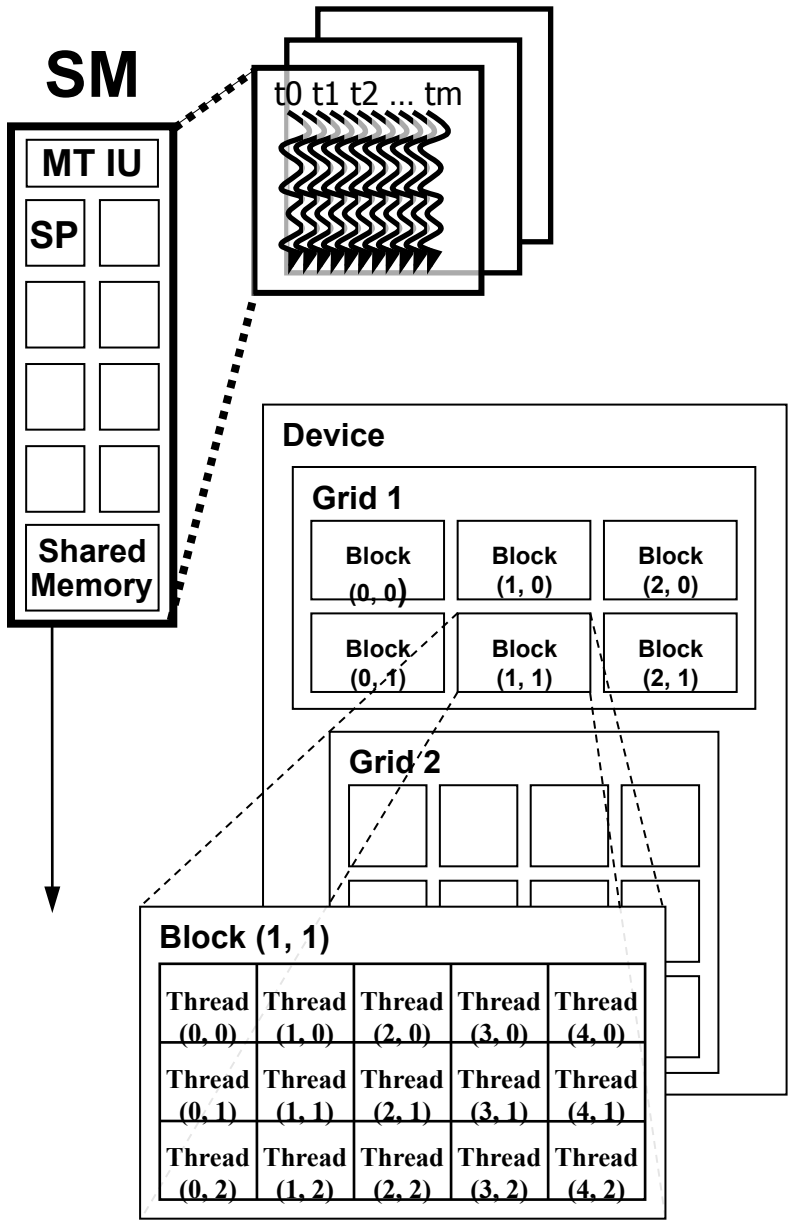


DavidKirk/NVIDIA & Wen-mei Hwu/UIUC



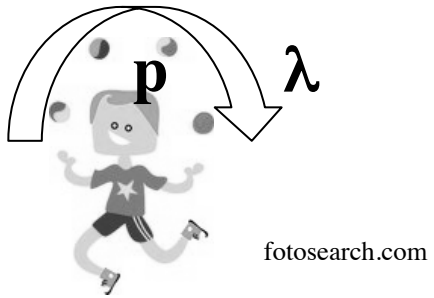
Constraints

- SM
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps
- All threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization
- Blocks
  - ≤ 512 threads
  - Registers subdivided over threads
- Synchronization among all threads in the block, only within block



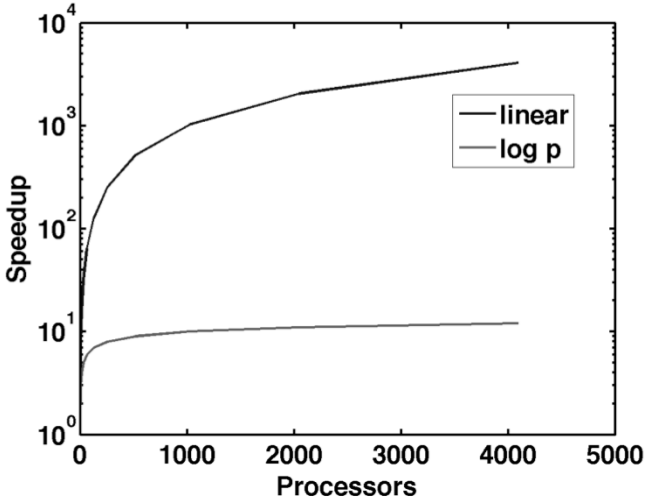
Occupancy

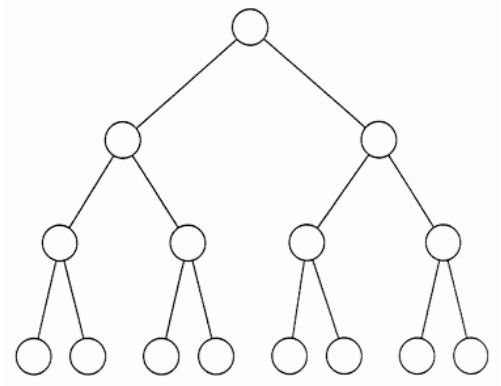
- Ratio: # active warps ÷ max # warps supported by vector unit
- Limited by vector units resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of thread
- Minimum # active warps required to hide latency
- Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 [38*256 =9728 < 16k]
  - # available registers is the limiting factor
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption



"Cartoon:" how to obtain good speedups

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
- Hide latency of host ↔ device memory transfers
- Reduce global memory accesses: use fast on-chip memories
- Manage occupancy
  - Registers and shared memory
- Coalesced memory transfers
- Avoid costly branches, or render them harmless



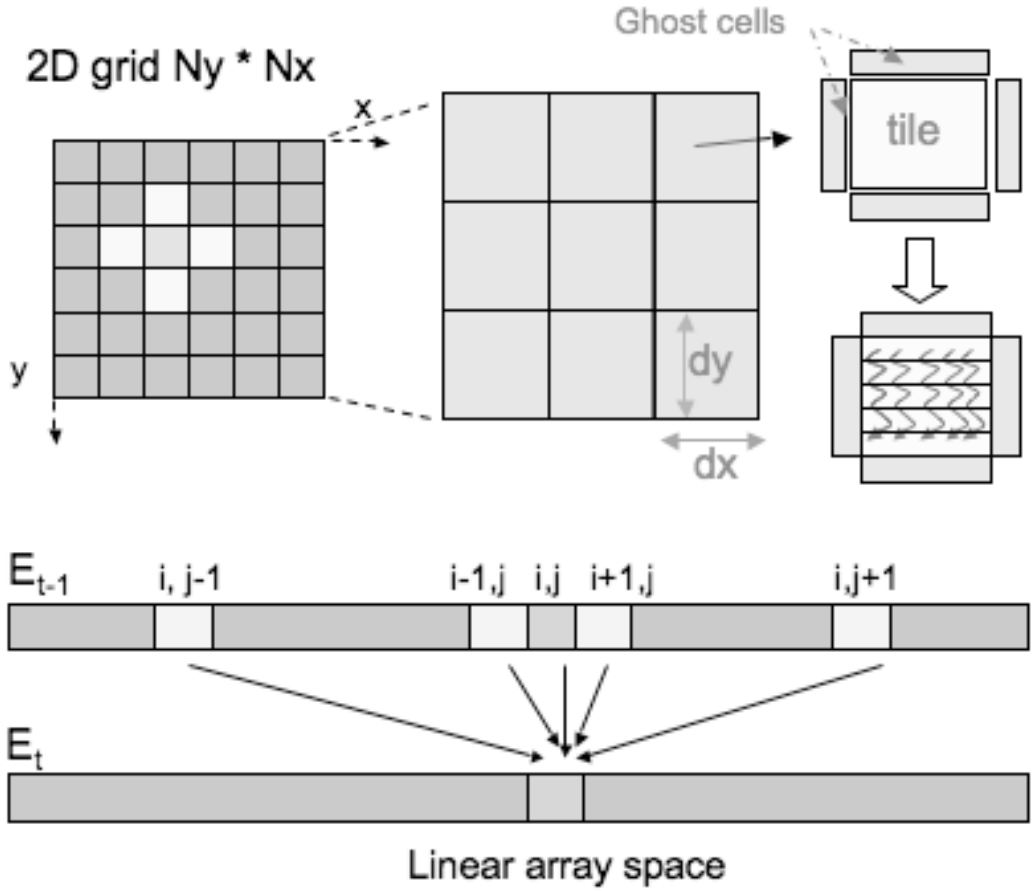


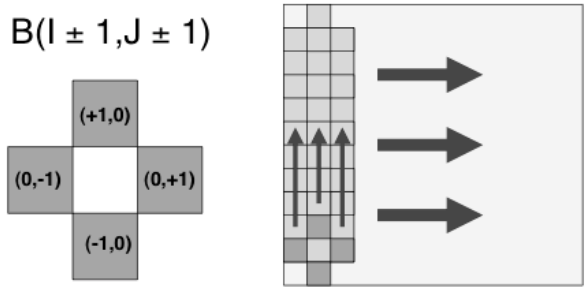
Is it that simple?

- Simplified processor design, but more user control over the hardware resources
  - Rethink problem solving technique & implementation
  - Redesign the software

- If we don't use the parallelism, we lose it
  - Amdahl's law - serial sections
  - Von Neumann bottleneck – data transfer costs
  - Workload Imbalances

Aliev-Panfilov implementation under CUDA





Rivera and Tseng

$$E^{t}[i,j] = E^{t-1}[i,j] + \alpha(E^{t-1}[i+1,j] + E^{t-1}[i-1,j] + E^{t-1}[i,j+1] + E^{t-1}[i,j-1] - 4E^{t-1}[i,j])$$

©2010 Scott B. Baden /CSE 260/ Fall '10

Naïve Aliev-Panfilov Code

- All array references go through device memory
- ./apf -n 6144 -t 0.04, 16x16 thread blocks
- Single Prec: 31 Gflops [Triton, 32 cores, MPI: 85GF]
- Double Prec: 17 Gflops (19GF n=8192) [Triton: 48 GF]

```
#define  E′ [i,j]  E_prev[(j+1)*(m+3) + (i+1)]
I =  blockIdx.y*blockDim.y + threadIdx.y;
J =  blockIdx.x*blockDim.x + threadIdx.x;
if ((I <= n) && (J <= m) )
   E[I] =E′ [I,J] + α*(E′[I-1,J] + E′[I+1,J] +E′[I,J-1]+E′ [I,J+1] -
                        4*E′ [I,J]);
for (j=1; j<= m+1; j++)
    for (i=1; i<= n+1; i++)
      E[j][i] = E′ [j][i]+α*(E′ [j][i-1]+E′ [j][i+1] +
                              E′ [j-1][i]+E′ [j+1][i] - 4*E′ [j][i]);
```

Optimized Aliev-Panfilov Code

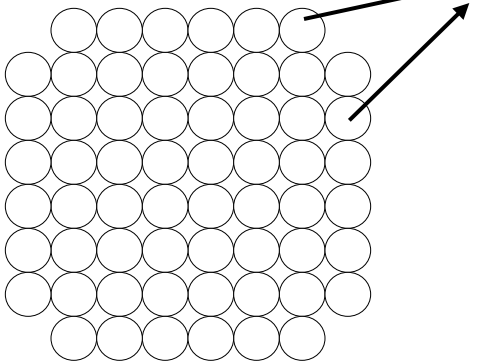
- Use shared memory:
  31 Gflops → 126 Gflops
  85 Gflops on 32 Triton cores with MPI

```
__shared__  float block[BDIM_Y + 2 ][BDIM_X + 2 ];
int idx = threadIdx.x, idy = threadIdx.y;
int y = blockIdx.y * (BDIM_Y) + idy;
int x = blockIdx.x * (BDIM_X) + idx ; idx++; idy++;
unsigned int index =  (y+1) * (m+3) +  (x+1)  ;

block[idy][idx] = center = E_prev[index];
__syncthreads();
e  = center + α*(block[idy][idx-1] + block[idy][idx+1] +
          block[idy-1][idx] + block[idy+1][idx] - 4*center);
e = e - dt*(kk * e * ( e- a) * ( e - 1 ) + e * r);
E[index] = e;
```

Ghost Cells

- Introduce branches and thread divergence

```
int idx = threadIdx.x, idy = threadIdx.y;
int y = blockIdx.y * (BDIM_Y) + idy++;
int x = blockIdx.x * (BDIM_X) + idx++;
unsigned int index =  y * N +  x
if(idy == 1 && y > 0 ){
//read a ghost cell into shared memory
block[0][idx] = E_prev [index-N];
```

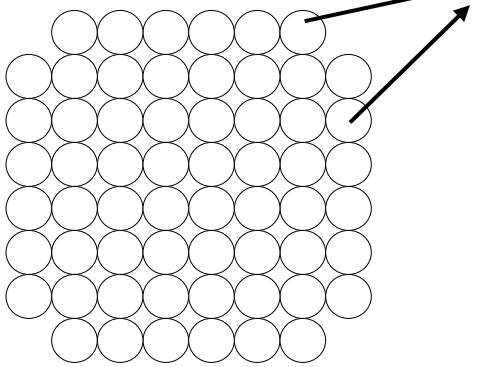


Ghost cells

When loading ghost cells, only some of the threads are active, some are idle

Ghost Cells (cont.)

- Divide the work between threads so that
  - each thread is responsible for 1 ghost cell load
- For a tile size of 16 x 16
  - There are 16x4 = 64 ghost cells
  - Then, create 64 threads



Ghost cells

Each thread will load one ghost cell

Synchronizzation

- void __syncthreads()
- Synchronizes **all** threads in a block
- Threads wait until all have arrived
- Avoids memory hazards
- Beware of conditionals
  if (condition(x)) __syncthreads();

©2010 Scott B. Baden /CSE 260/ Fall '10

Reduction

Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- Divergence when predicates are a function of the threadId

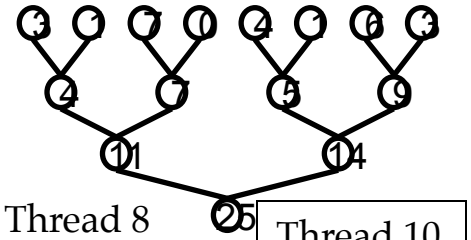
```
if (threadId < 2) { }
```

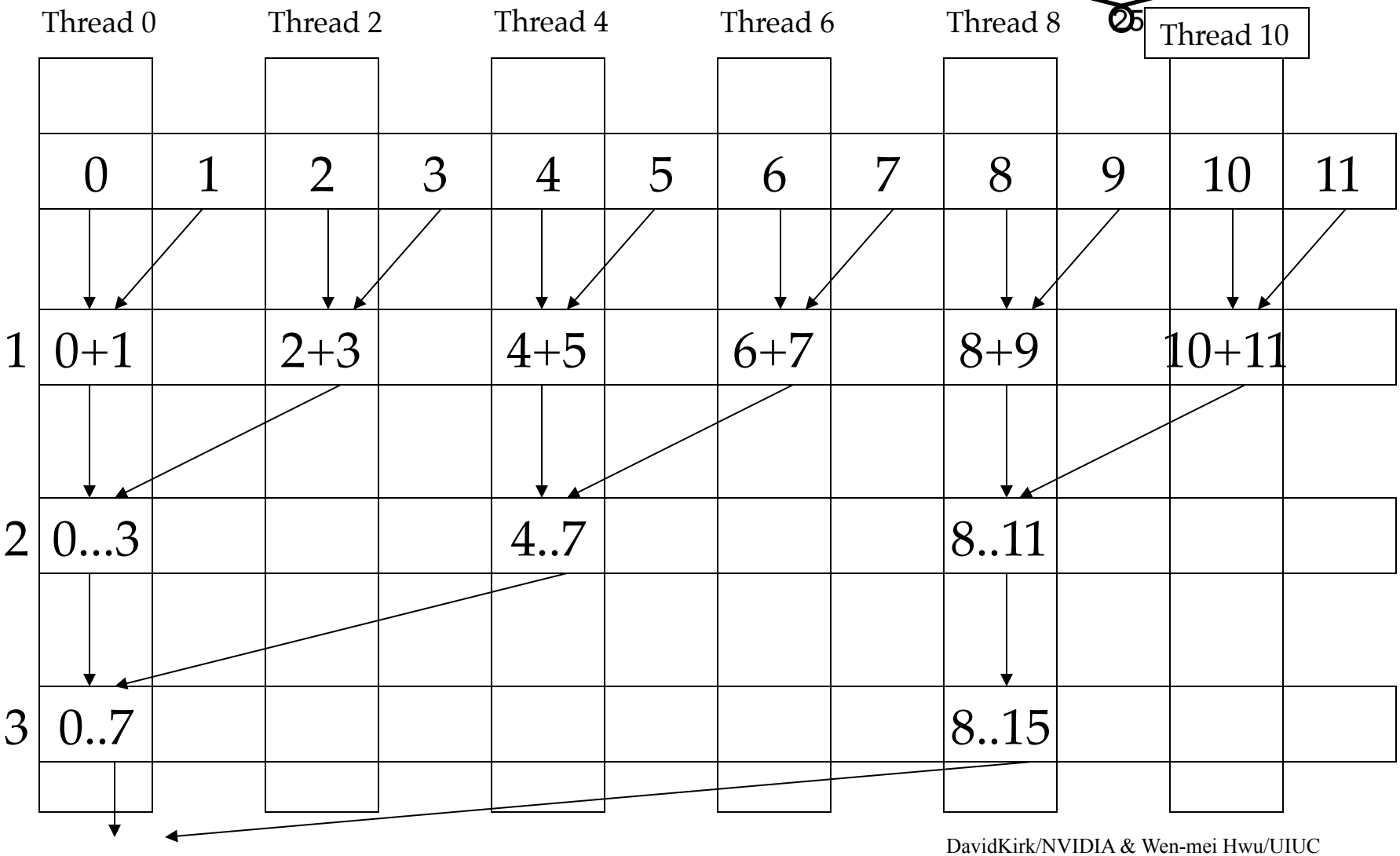
- We can avoid divergence, everyone executes the same path

```
if (threadId / WARP_SIZE < 2) { }
```

- Consider reduction, e.g. sumation $\sum_i x_i$

A naïve reduction





DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

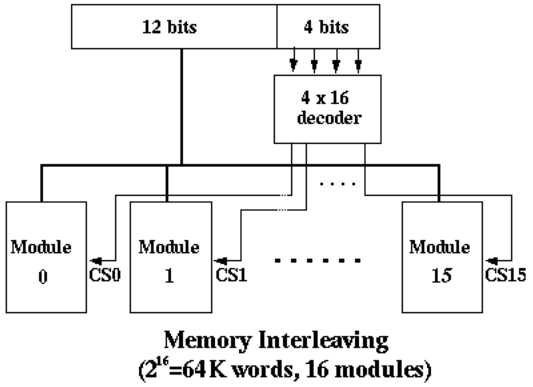
The naïve code

```
__global__ void reduce(int *input, unsigned int N, int *total){
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
   __shared__ int x[BSIZE];
   x[tid] = (i<N) ? input[i] : 0;
   __syncthreads();

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

if (tid == 0) atomicAdd(total,x[tid]);

}
```

Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

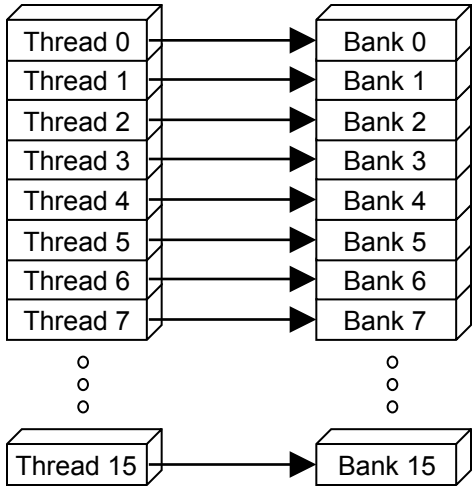


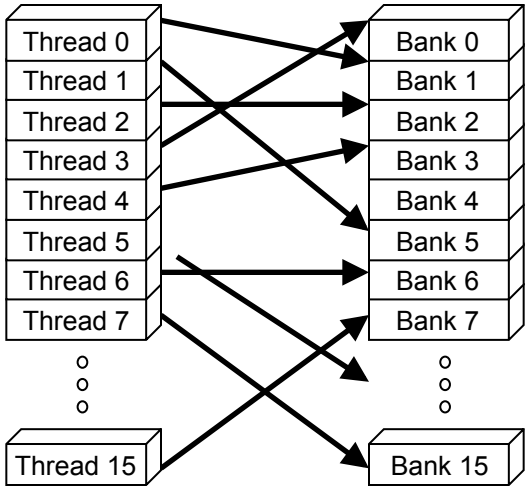
Ruye Wang,
fourier.eng.hmc.edu/e85/lectures/memory

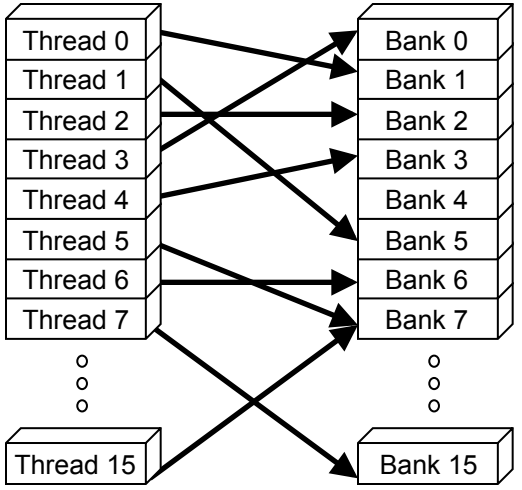
©2010 Scott B. Baden /CSE 260/ Fall '10

Bank conflicts

- Access to shared memory “as fast” as registers unless...
- 2 or more instructions in a 1/2 warp access different banks: we have a *conflict*
- Exception: if all access the same bank: broadcast
- Each bank can service 1 address / cycle (bcast, too)







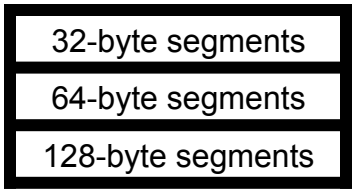

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.0f;
```

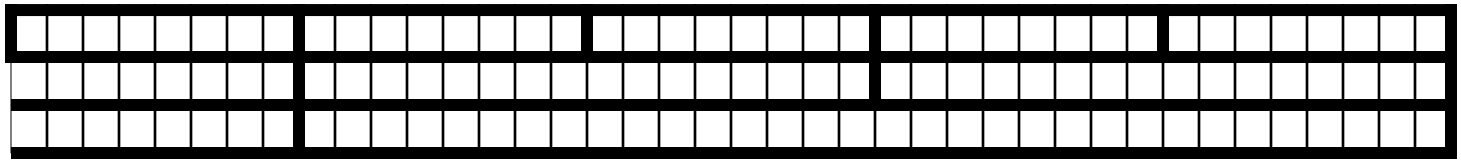
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Memory coalescing

- Memory banks: consecutive addresses can be read very quickly (K=0; Q=1)
- Certain non-sequential access patterns to global memory degrade performance K mod 16 ≠ 0; Q≠1)
- Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)

```
tid = blockIdx.x*blockDim.x+threadIdx.x + K
shared[tid] = global[tid]
int tid =(blockIdx.x*blockDim.x+ threadIdx.x)*Q
```



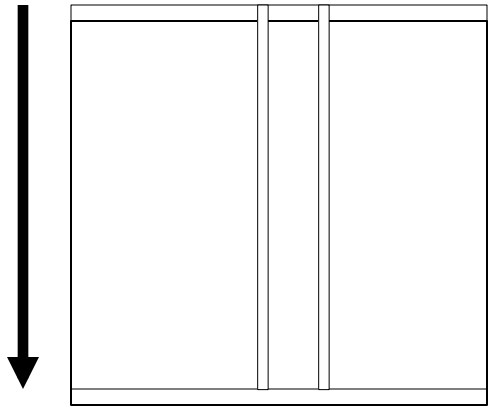


Nvidia Corp.

Memory coalescing

- Simplest: addresses are contiguous across threads
- Accesses organized by half warps (16 threads)

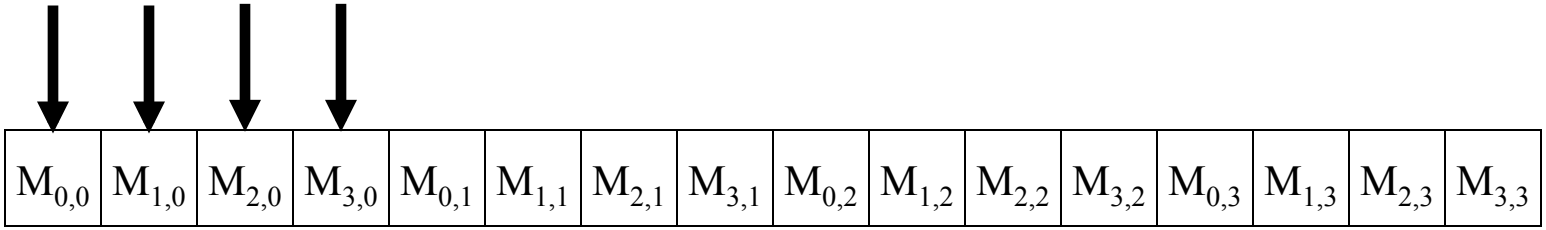
```
__shared__ a[ ][ ], b[ ][ ];
I =  blockIdx.y*by + ty;
J =  blockIdx.x*bx + tx;
a[ty][tx] = A[I*N+k*by+tx];
b[ty][tx] = B[J+N*(k*bx+ty)]
```



Threads →

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Tile iteration



Nvidia Corp.

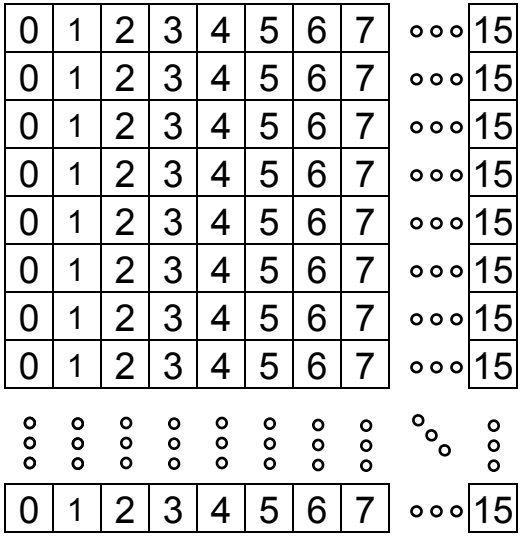
©2010 Scott B. Baden /CSE 260/ Fall '10

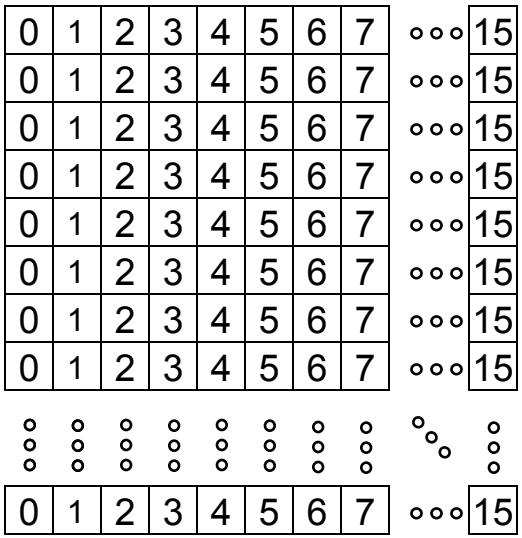
Bank conflicts in 2d arrays

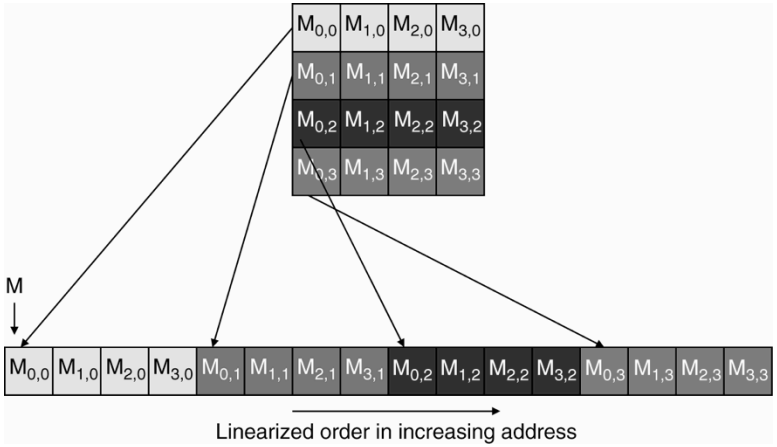
- All warps in a block access consecutive elements within a row as they step through neighboring columns (matrix multiply)

```
a[ty][tx] = A[I*N+k*by+tx]
b[ty][tx] = B[J+N*(k*bx+ty)];
```

- Accesses by threads in a block along a column don't coalesce

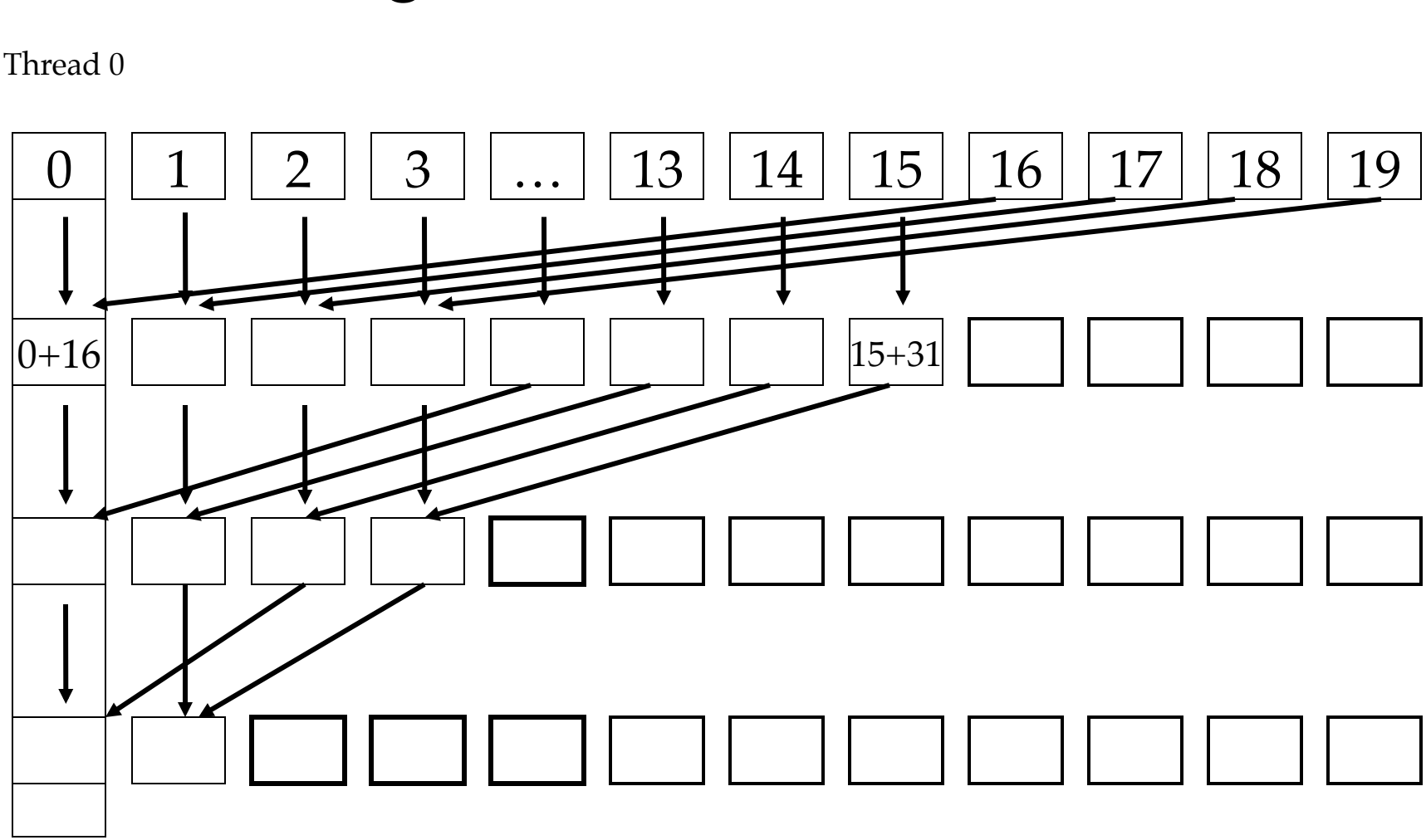






DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Reducing divergence and avoiding bank conflicts



DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

The Code

```
unsigned int tid = threadIdx.x;
 __shared__ int x[BSIZE];

…

 for (unsigned int s= blockDim.x/2; s>1;  s /= 2)
  {
     __syncthreads();

     if (tid < s)
          x[tid] += x[tid + s];

  }
```

Fin