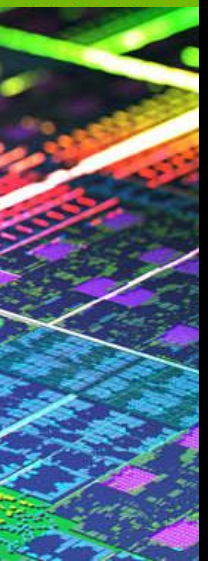
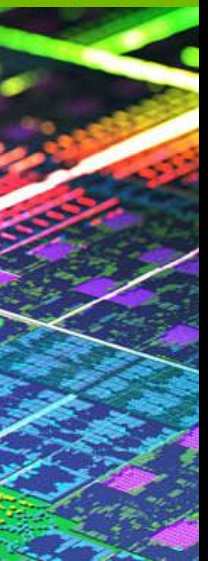
Shuffle: Tips and Tricks

Julien Demouth, NVIDIA



Glossary

- Warp
  - Implicitly synchronized group of threads (32 on current HW)
- Warp ID (`warpid`)
  - Identifier of the warp in a block: `threadIdx.x / 32`
- Lane ID (`laneid`)
  - Coordinate of the thread in a warp: `threadIdx.x % 32`
  - Special register (available from PTX): `%laneid`

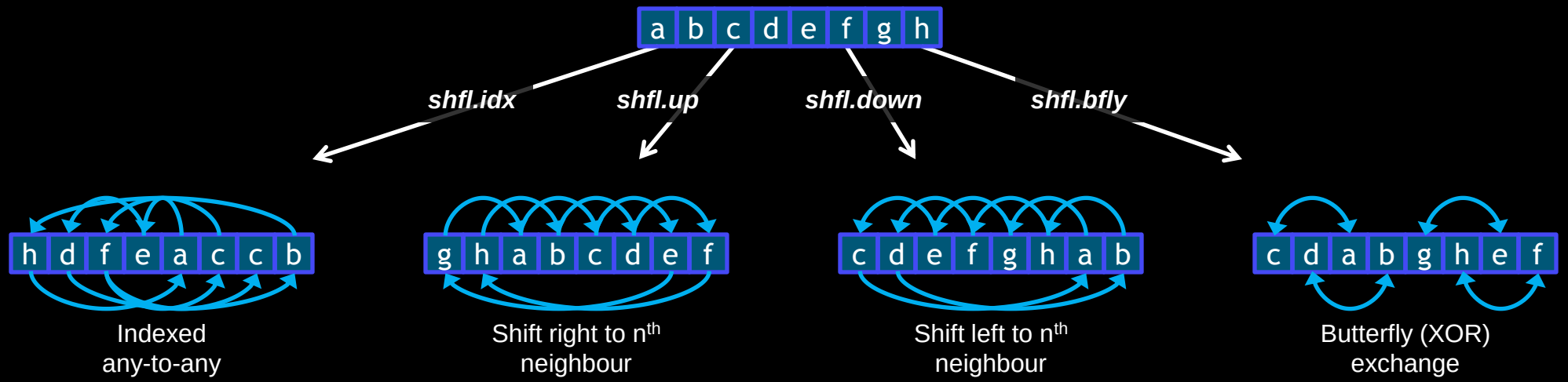


Shuffle (SHFL)

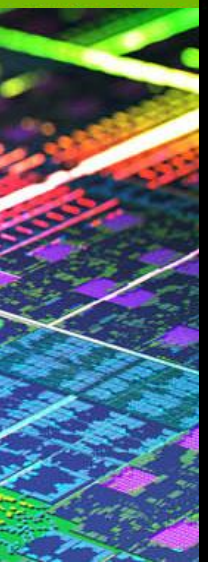
- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0

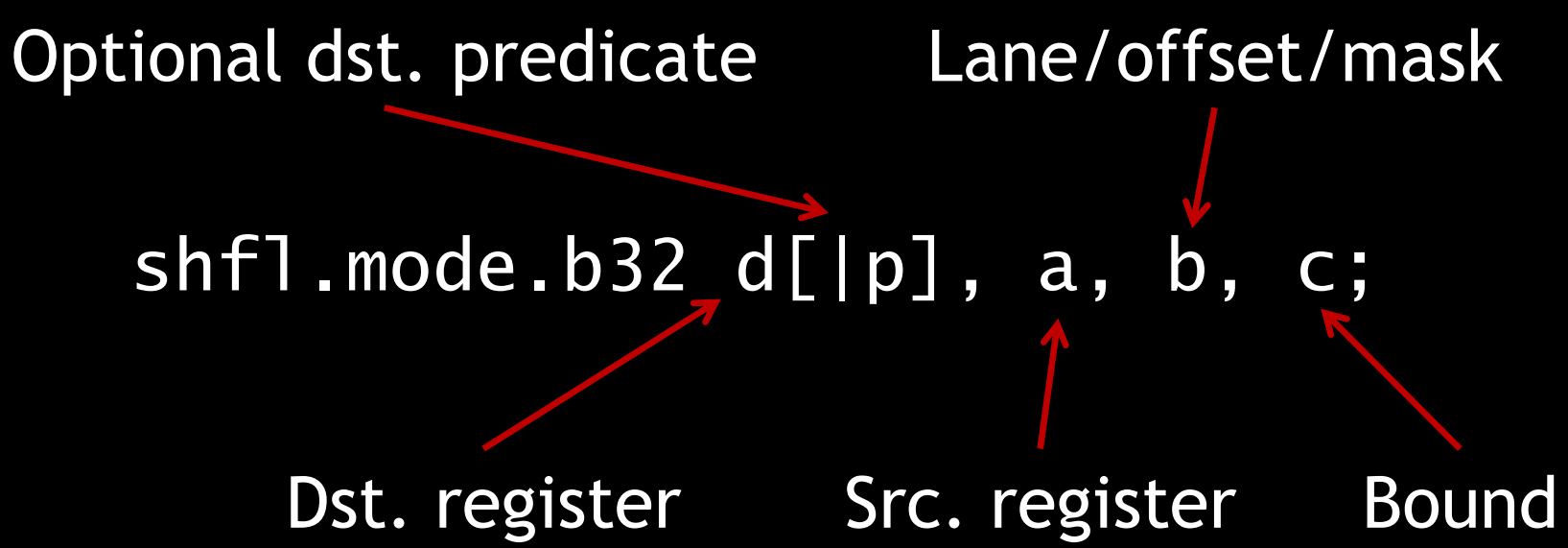
Variants

- 4 variants (idx, up, down, bfly):



Instruction (PTX)





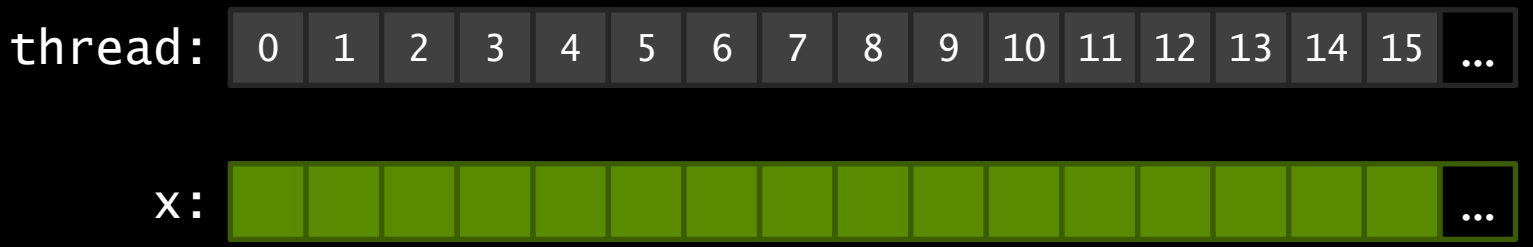
Implement SHFL for 64b Numbers

```
__device__ __inline__ double shfl(double x, int lane)
{
    // Split the double number into 2 32b registers.
    int lo, hi;
    asm volatile( “mov.b32 {%0,%1}, %2;” : “=r”(lo), “=r”(hi) : “d”(x));

    // Shuffle the two 32b registers.
    lo = __shfl(lo, lane);
    hi = __shfl(hi, lane);

    // Recreate the 64b number.
    asm volatile( “mov.b64 %0, {%1,%2};” : “=d(x)” : “r”(lo), “r”(hi));

    return x;
}
```

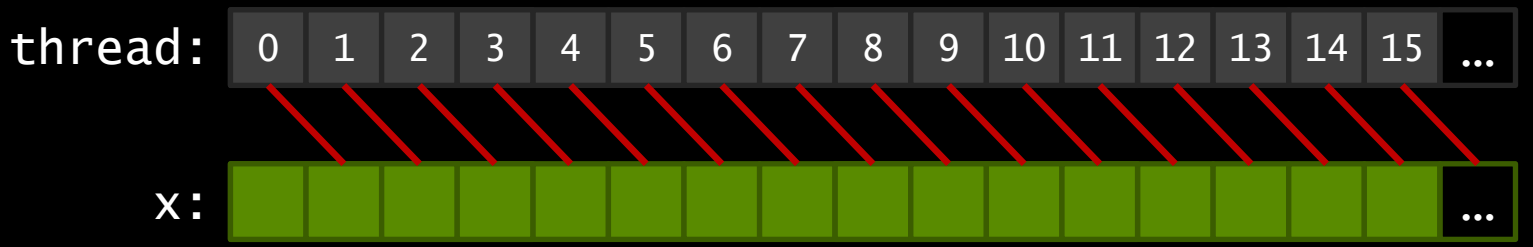
- Generic SHFL: https://github.com/BryanCatanzaro/generics

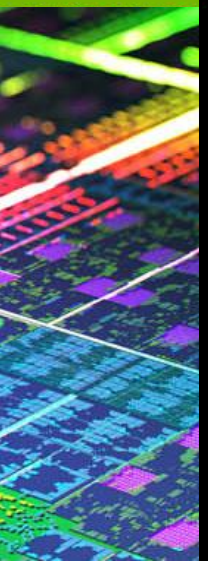
Performance Experiment

- One element per thread



- Each thread takes its right neighbor





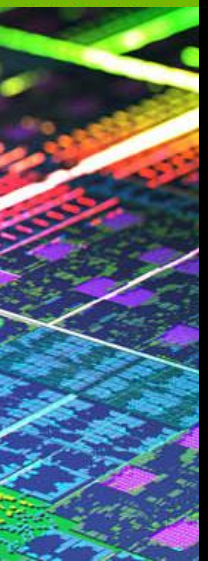
Performance Experiment

- We run the following test on a K20

```
T x = input[tidx];
for(int i = 0 ; i < 4096 ; ++i)
    x = get_right_neighbor(x);
output[tidx] = x;
```

- We launch 26 blocks of 1024 threads
  - On K20, we have 13 SMs
  - We need 2048 threads per SM to have 100% of occupancy

- We time different variants of that kernel



Performance Experiment

- Shared memory (SMEM)

```
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];

__syncthreads();
```

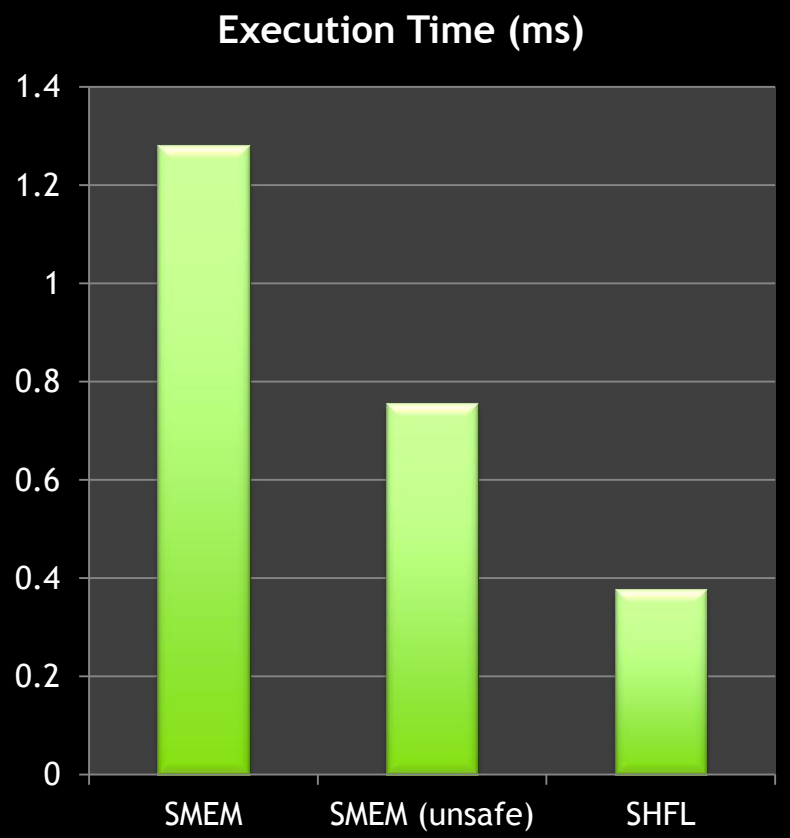
- Shuffle (SHFL)

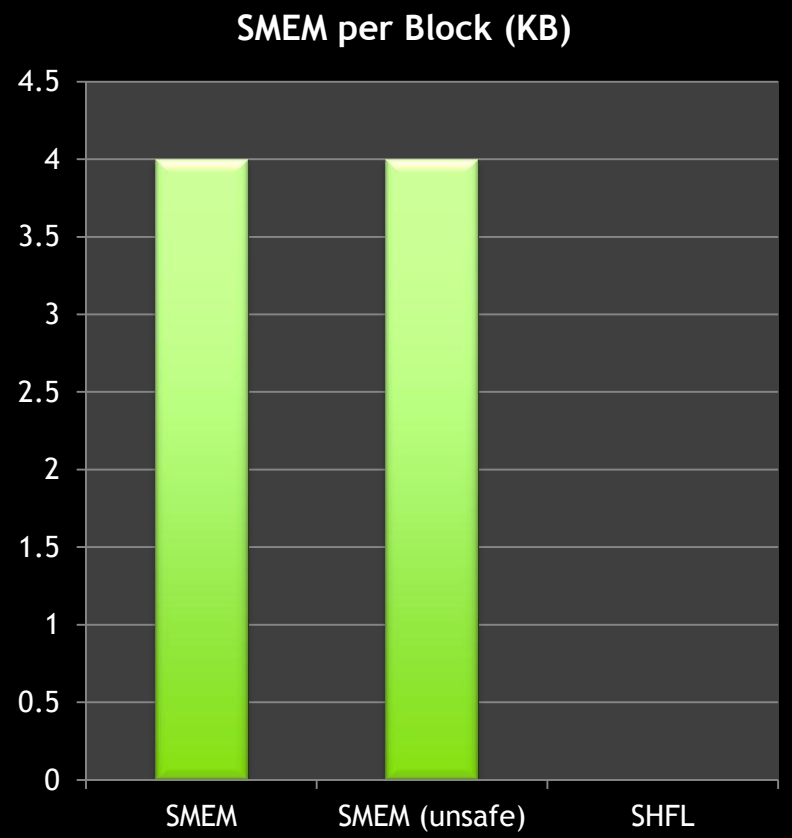
```
x = __shfl(x, (laneid+1) % 32);
```

- Shared memory without __syncthreads + volatile (*unsafe*)

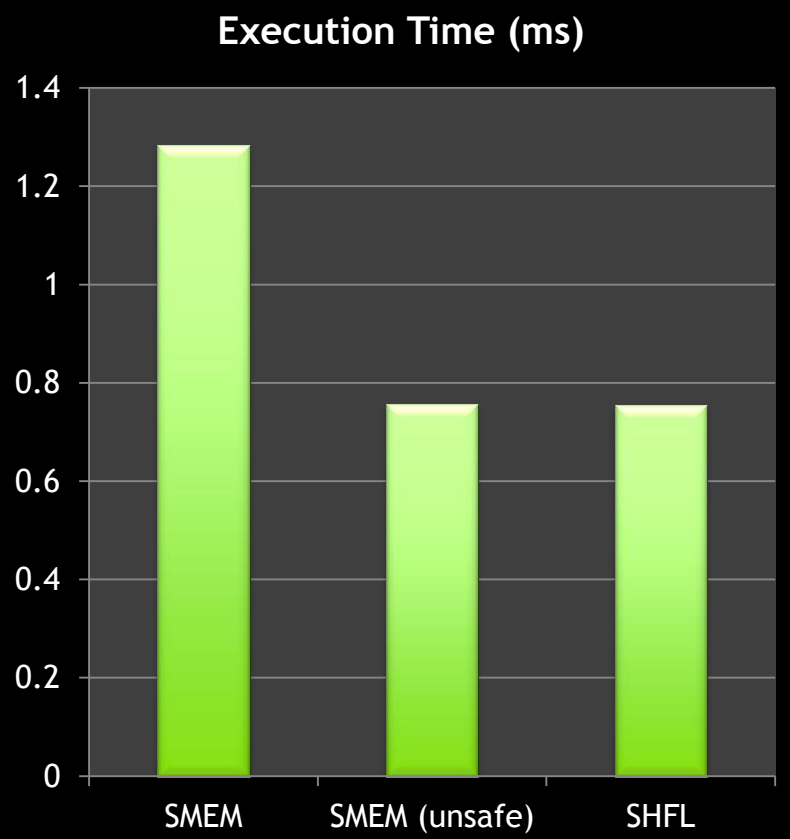
```
__shared__ volatile T *smem = ...;

smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];
```

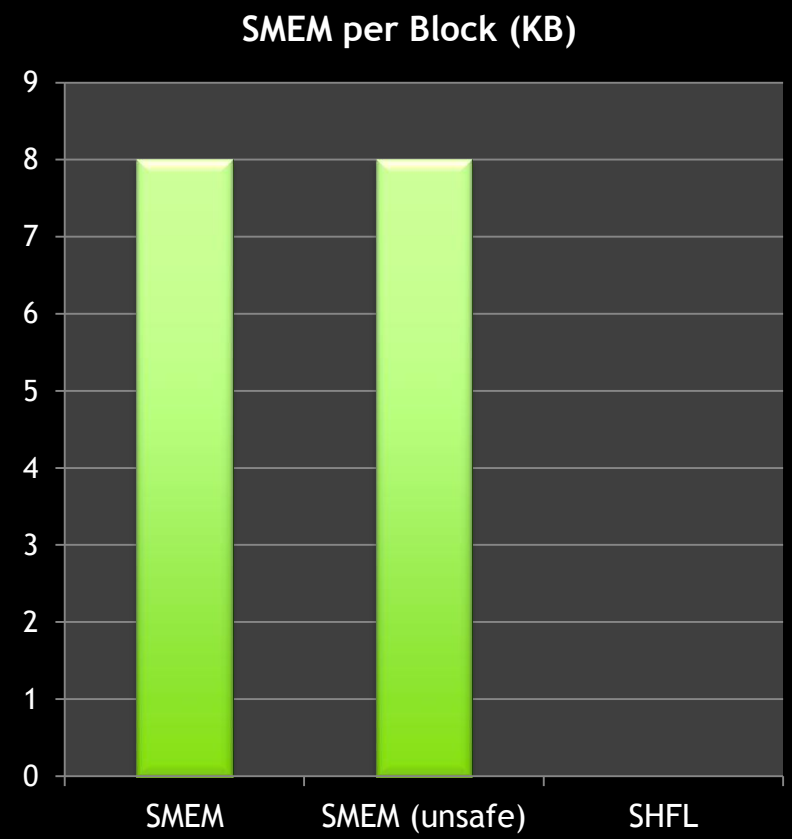
Performance Experiment (fp32)

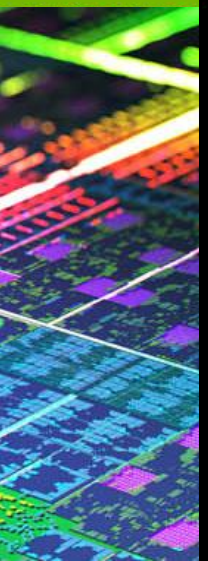




Performance Experiment (fp64)

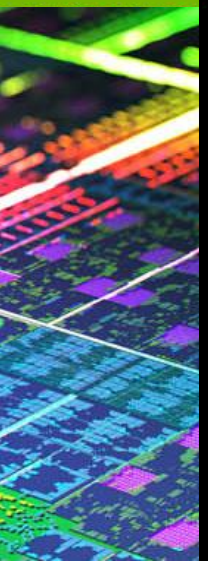






Performance Experiment

- Always faster than shared memory
- Much safer than using no __syncthreads (and volatile)
  - And never slower
- Does not require shared memory
  - Useful when occupancy is limited by SMEM usage



Broadcast

- All threads read from a single lane

```
x = __shfl(x, 0); // All the threads read x from laneid 0.
```

- More complex example

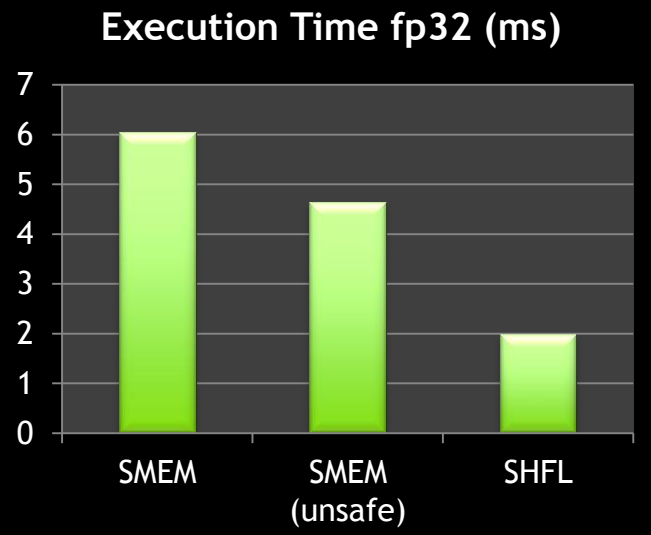
```
// All threads evaluate a predicate.
int predicate = ...;

// All threads vote.
unsigned vote = __ballot(predicate);

// All threads get x from the “last” lane which evaluated the predicate to true.
if(vote)
    x = __shfl(x, __bfind(vote));


// __bind(unsigned i): Find the most significant bit in a 32/64 number (PTX).
__bfind(&b, i) { asm volatile(“bfind.u32 %0, %1;” : “=r”(b) : “r”(i)); }
```

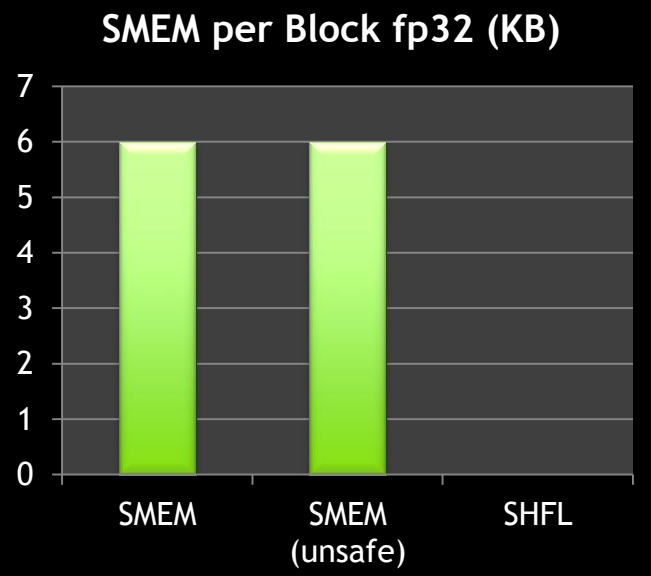
Reduce

- Code

```
// Threads want to reduce the value in x.

float x = …;

#pragma unroll
for(int mask = WARP_SIZE / 2 ; mask > 0 ; mask >>= 1)
    x += __shfl_xor(x, mask);

// The x variable of laneid 0 contains the reduction.
```

- Performance
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times



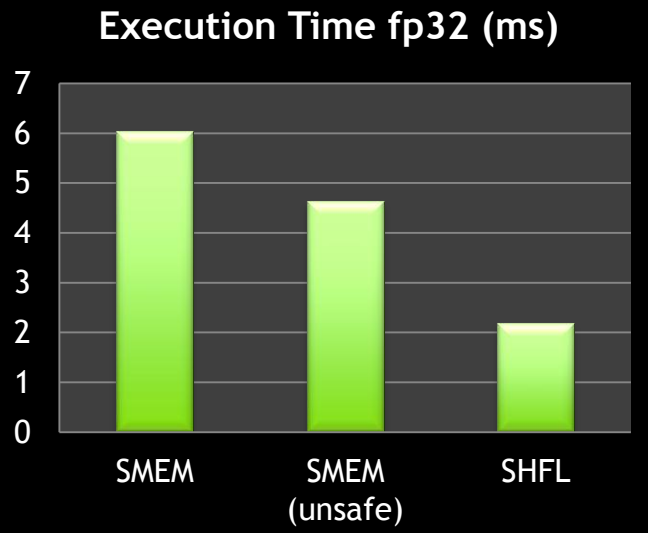


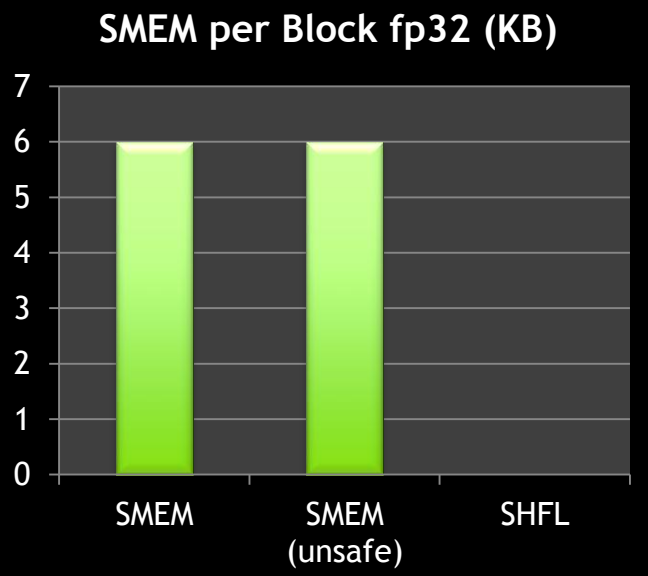
Scan

- Code

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <<= 1 )
{
    float y = __shfl_up(x, offset);
    if(laneid() >= offset)
        x += y;
}
```

- Performance
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times



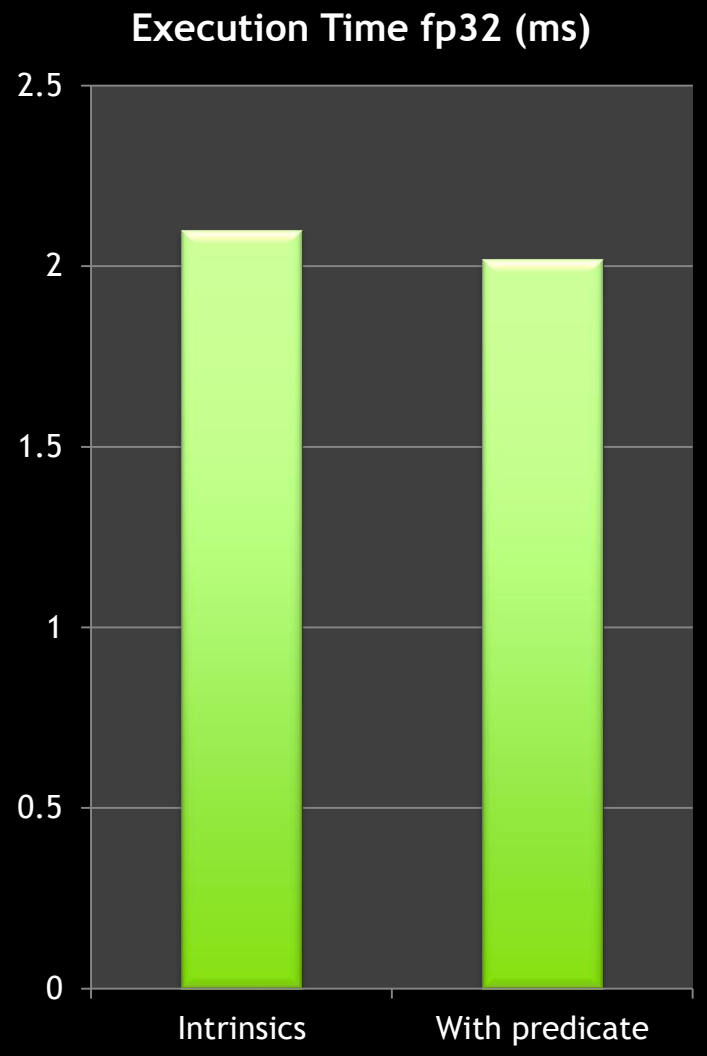


Scan

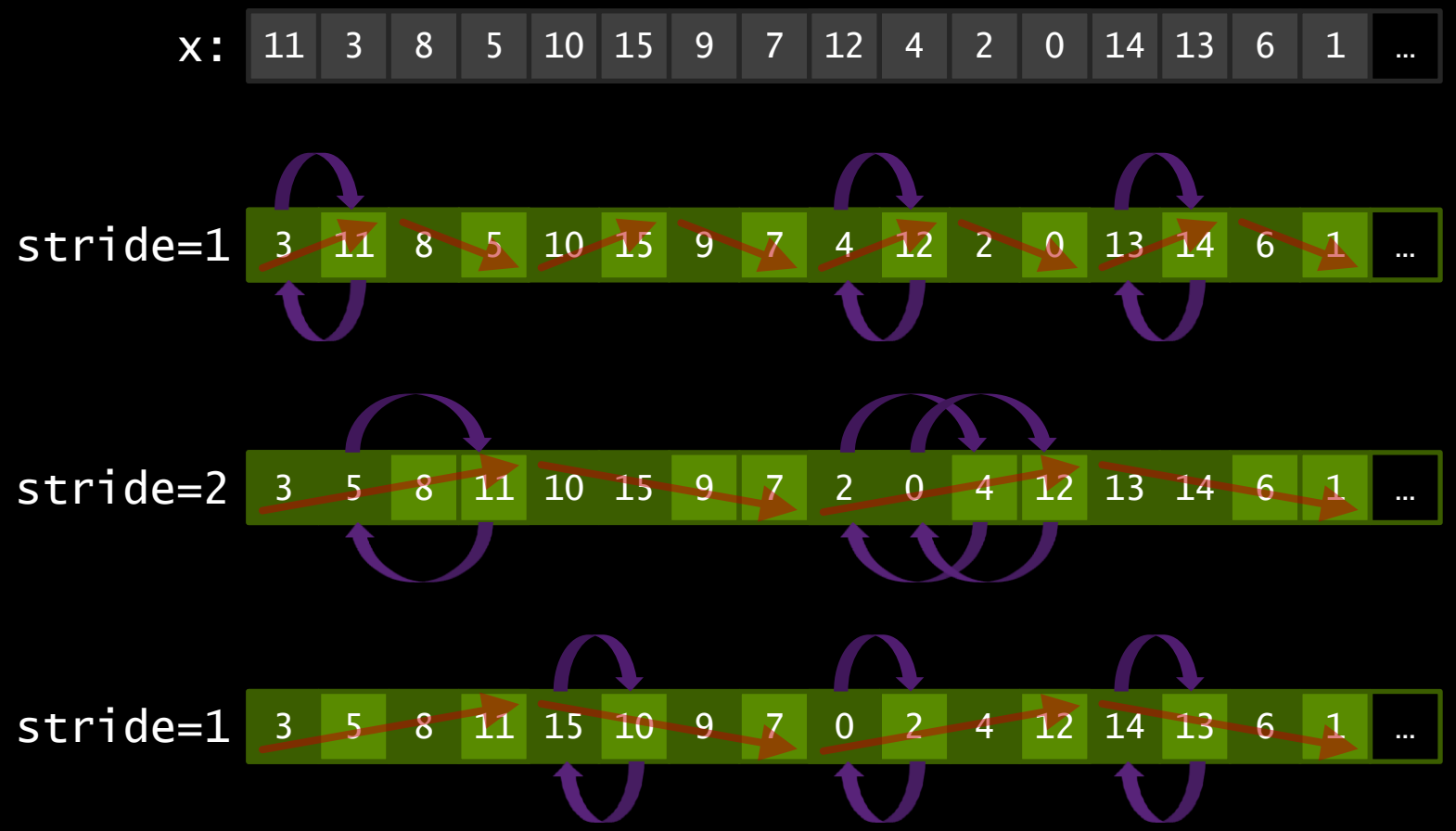
- Use the predicate from SHFL

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <<= 1 )
{
    asm volatile( "{"
          "  .reg .f32 r0;"
          "  .reg .pred p;"
          "  shfl.up.b32 r0|p, %0, %1, 0x0;"
          "  @p add.f32 r0, r0, %0;"
          "  mov.f32 %0, r0;"
          "}“ : "+f"(x) : "r"(offset));
}
```

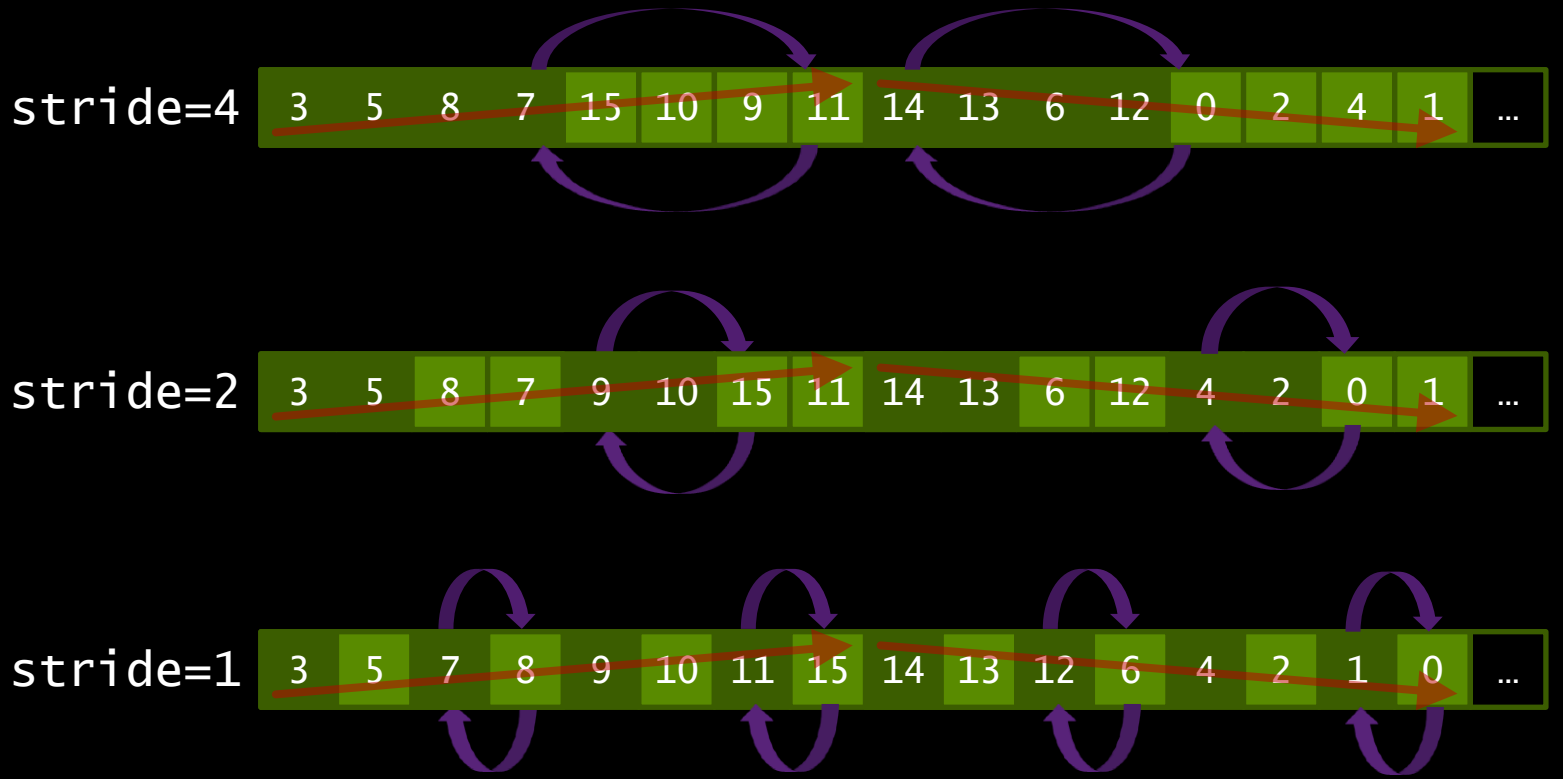
- Use CUB:
  https://nvlabs.github.com/cub



Bitonic Sort

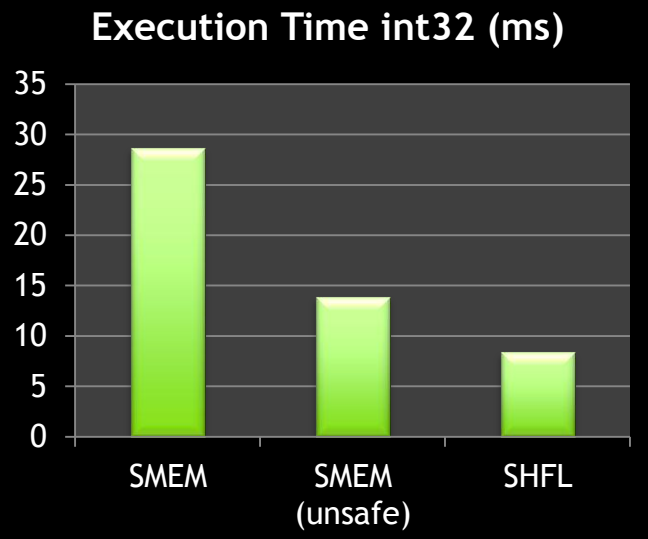


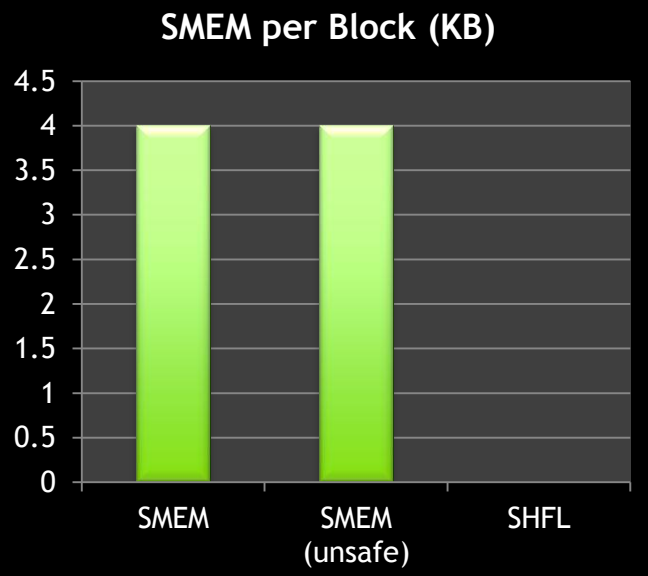
Bitonic Sort



Bitonic Sort

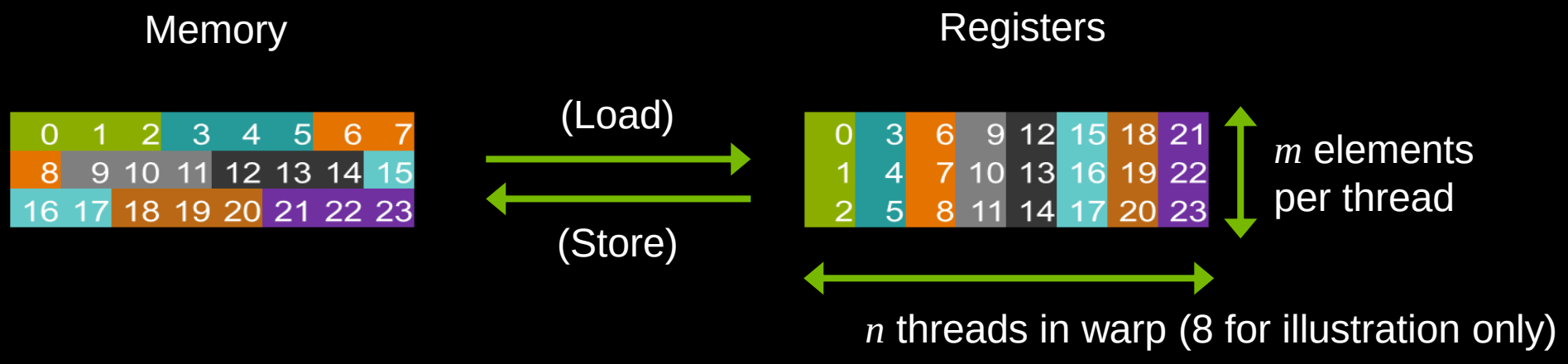
```
int swap(int x, int mask, int dir)
{
     int y = __shfl_xor(x, mask);
     return x < y == dir ? y : x;
}

x = swap(x, 0x01, bfe(laneid, 1) ^ bfe(laneid, 0)); //  2
x = swap(x, 0x02, bfe(laneid, 2) ^ bfe(laneid, 1)); //  4
x = swap(x, 0x01, bfe(laneid, 2) ^ bfe(laneid, 0));
x = swap(x, 0x04, bfe(laneid, 3) ^ bfe(laneid, 2)); //  8
x = swap(x, 0x02, bfe(laneid, 3) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 3) ^ bfe(laneid, 0));
x = swap(x, 0x08, bfe(laneid, 4) ^ bfe(laneid, 3)); // 16
x = swap(x, 0x04, bfe(laneid, 4) ^ bfe(laneid, 2));
x = swap(x, 0x02, bfe(laneid, 4) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 4) ^ bfe(laneid, 0));
x = swap(x, 0x10,                  bfe(laneid, 4)); // 32
x = swap(x, 0x08,                  bfe(laneid, 3));
x = swap(x, 0x04,                  bfe(laneid, 2));
x = swap(x, 0x02,                  bfe(laneid, 1));
x = swap(x, 0x01,                  bfe(laneid, 0));

// int bfe(int i, int k): Extract k-th bit from i

// PTX: bfe dst, src, start, len (see p.81, ptx_isa_3.1)
```





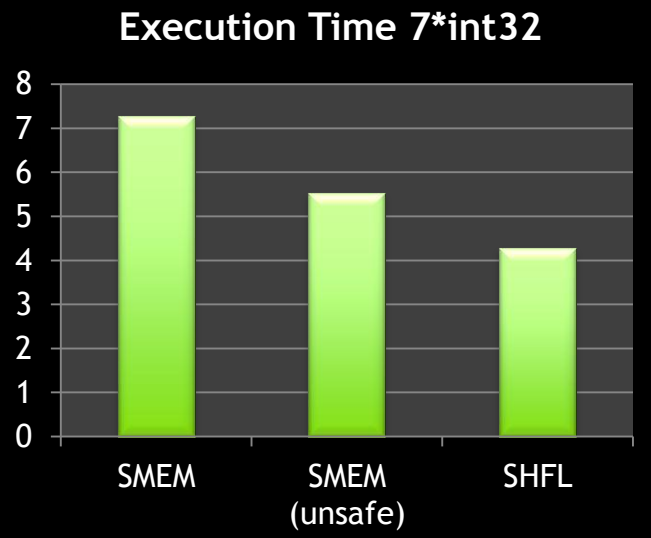
Transpose

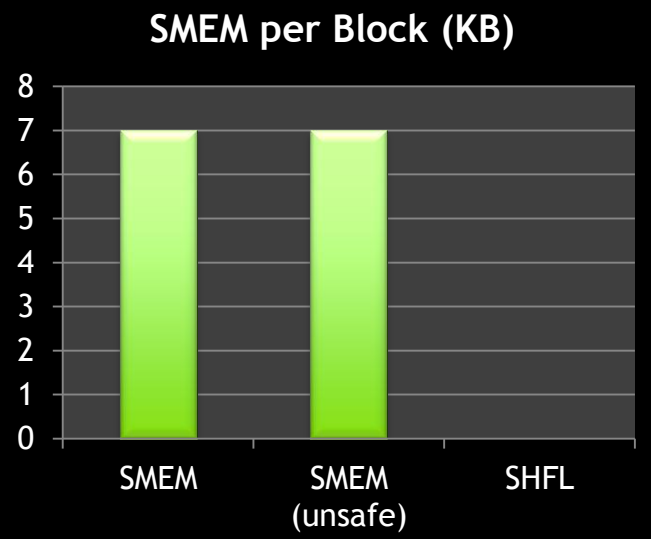
- When threads load or store arrays of structures, transposes enable fully coalesced memory operations
- *e.g.* when loading, have the warp perform coalesced loads, then transpose to send the data to the appropriate thread



Transpose

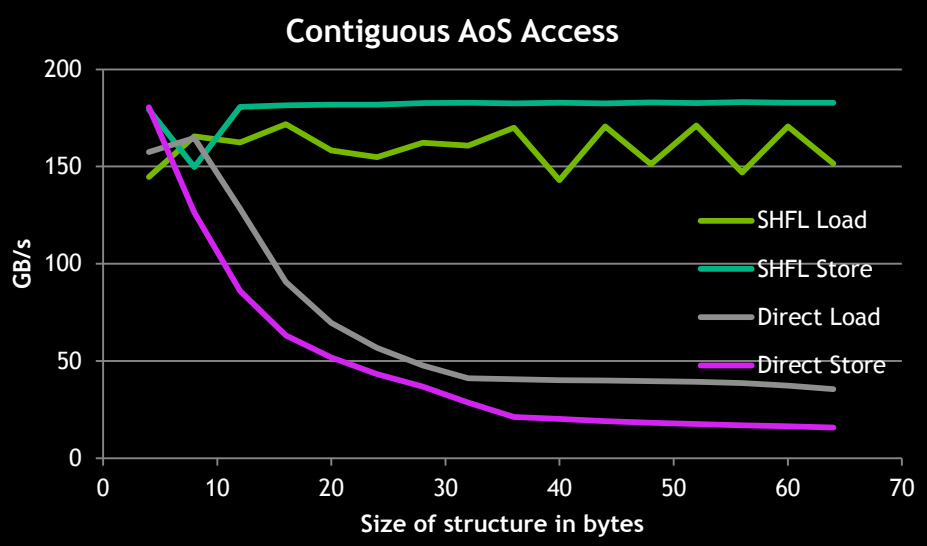
- You can use SMEM to implement this transpose, or you can use SHFL
- Code:
  http://github.com/bryancatanzaro/trove
- Performance
  - Launch 104 blocks of 256 threads
  - Run the transpose 4096 times

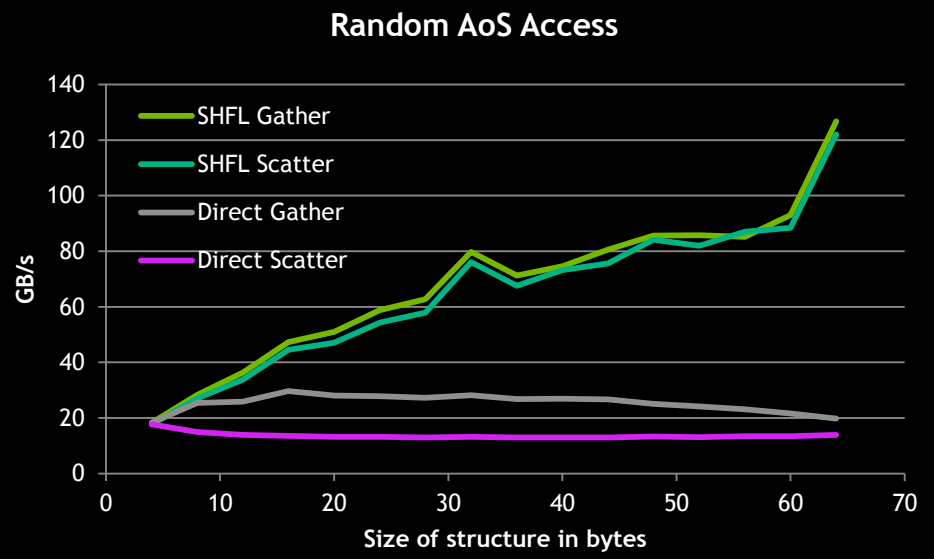


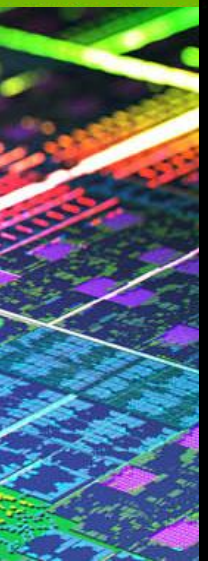


Array of Structures Access via Transpose

- Transpose speeds access to arrays of structures
- High-level interface: `coalesced_ptr<T>`
  - Just dereference like any pointer
  - Up to 6x faster than direct compiler generated access







Conclusion

- SHFL is available for SM >= SM 3.0
- It is always faster than “safe” shared memory
- It is never slower than “unsafe” shared memory
- It can be used in many different algorithms