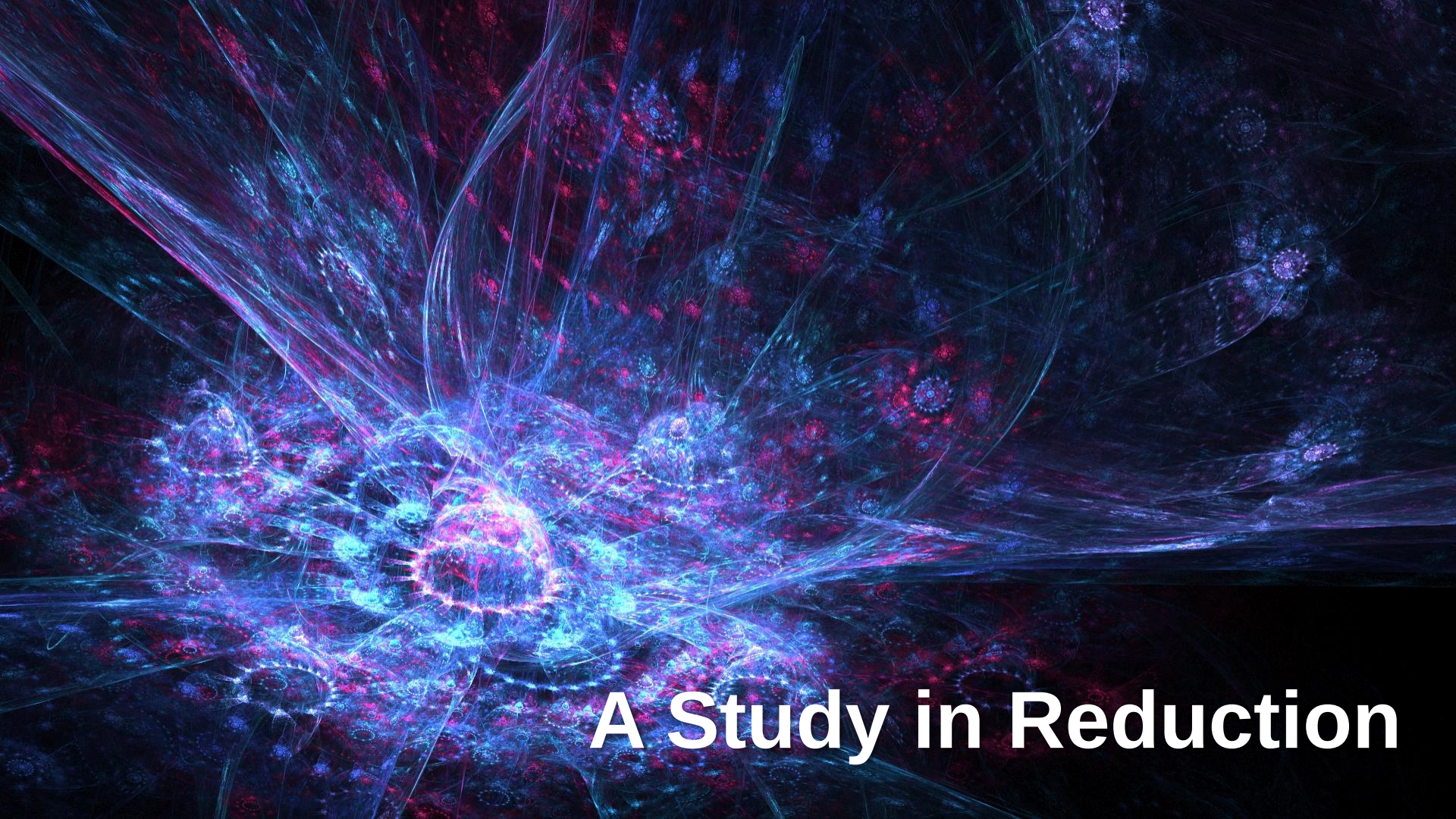
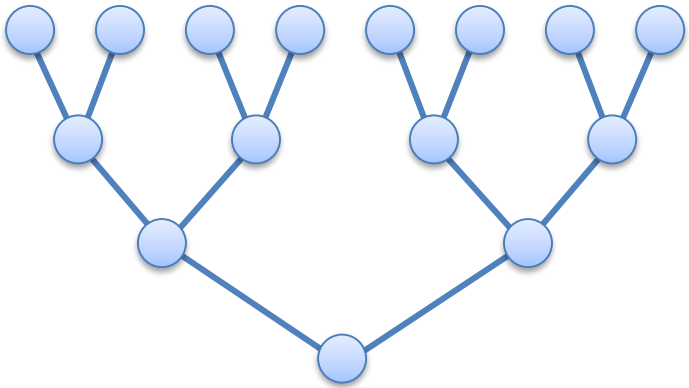

A Study in Reduction

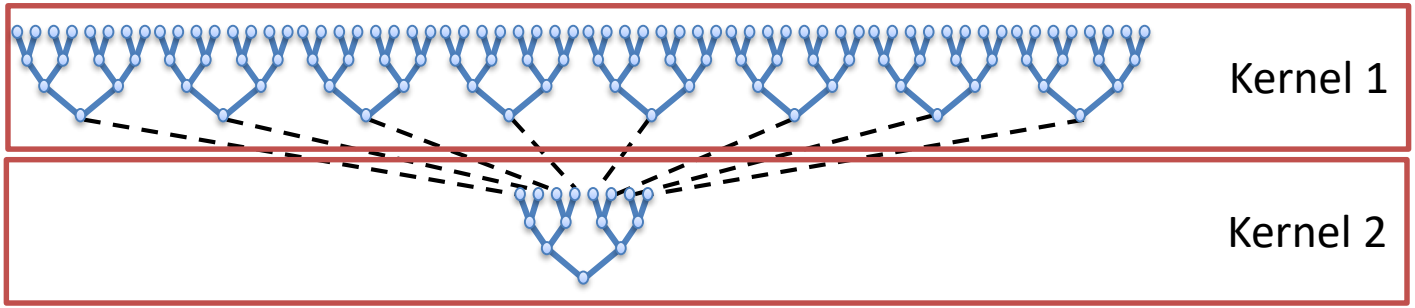
Parallel Reduction

- Common and important data parallel primitive
- Many data elements -> single output (associative!)
- Easy to implement in CUDA
- Tree-based approach



Parallel Reduction

- Need to use multiple thread blocks
  - To process large arrays
  - To fully utilize GPU
- Partition the array, one block per partition
- How to communicate partial results?



Reduction #1: AtomicGlobal

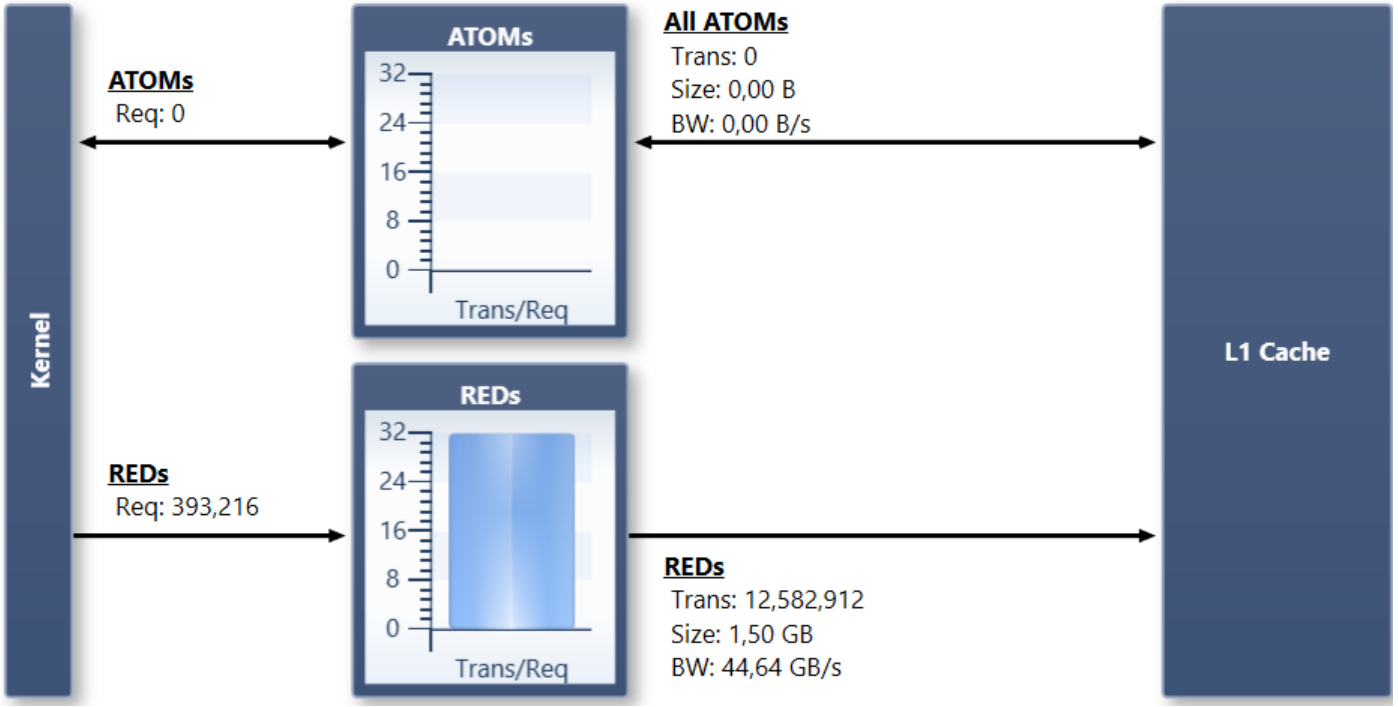
```
__global__ void reduceAtomicGlobal(const float* input,
           float* result, int elements)
{
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  atomicAdd(result, input[id]);
}
```

Performance 12M Reduction

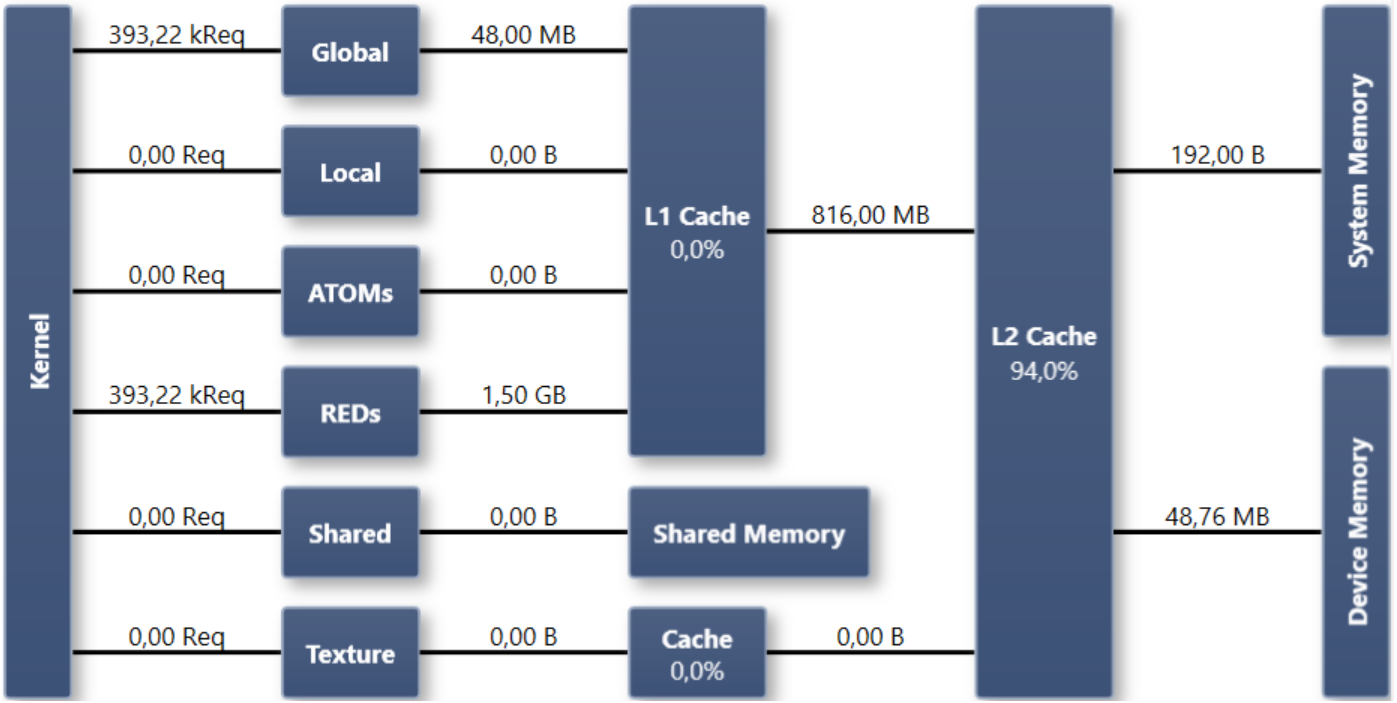
	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	

GTX 680 peak: 192GB/S

Reduction #1: AtomicGlobal



Reduction #1: AtomicGlobal



Reduction #2: AtomicShared

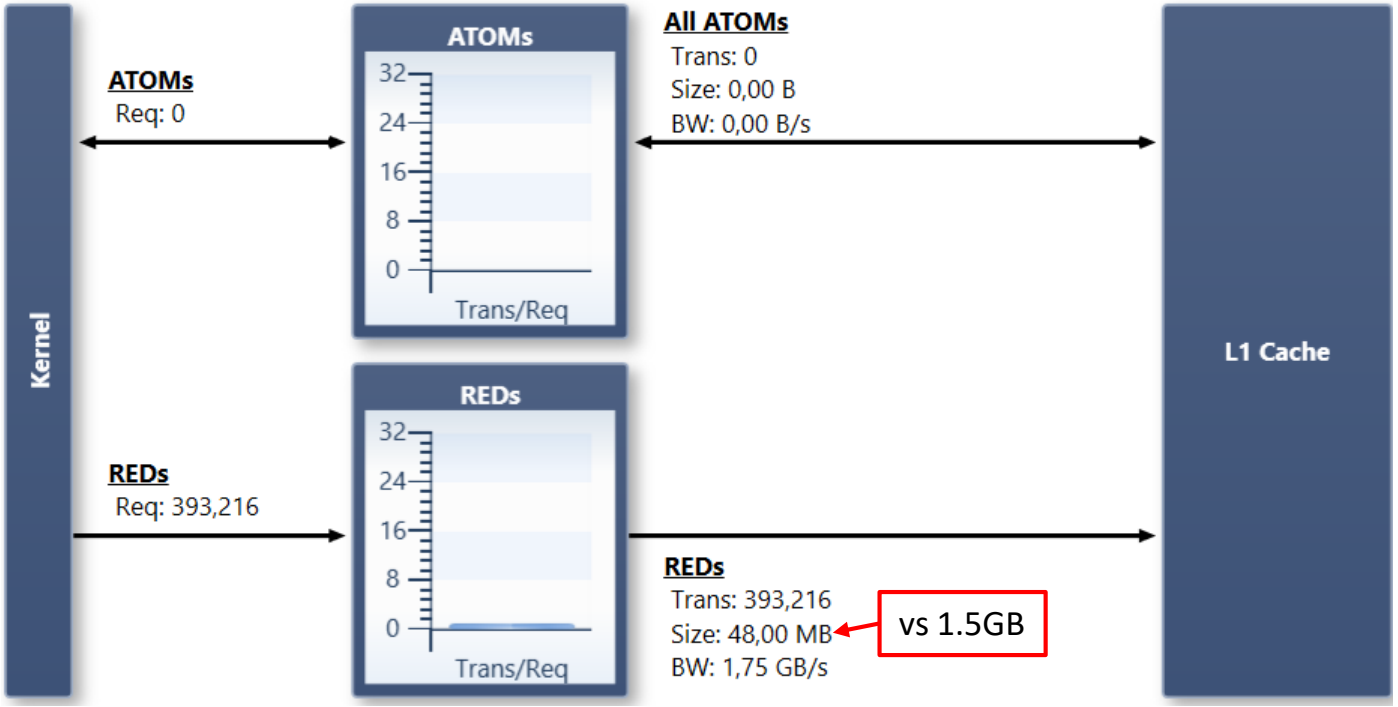
```
__global__ void reduceAtomicShared(const float* input,
           float* result, int elements)
{
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  float in = input[id];

  __shared__ float x;
  x = 0.0f;
  __syncthreads();

  atomicAdd(&x, in);
  __syncthreads();

  if(threadIdx.x == 0)
    atomicAdd(result, x);
}
```

Performance 12M Reduction

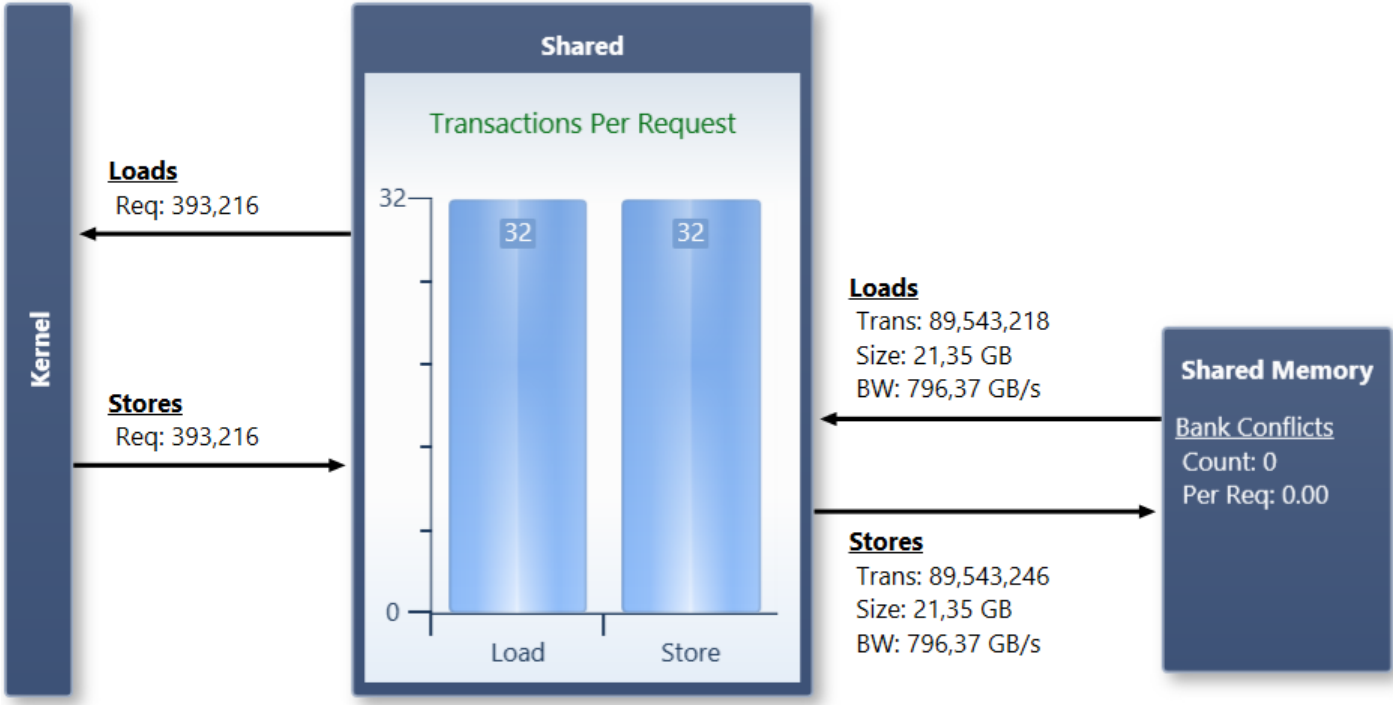
	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	

GTX 680 peak: 192GB/S

Reduction #2: AtomicShared



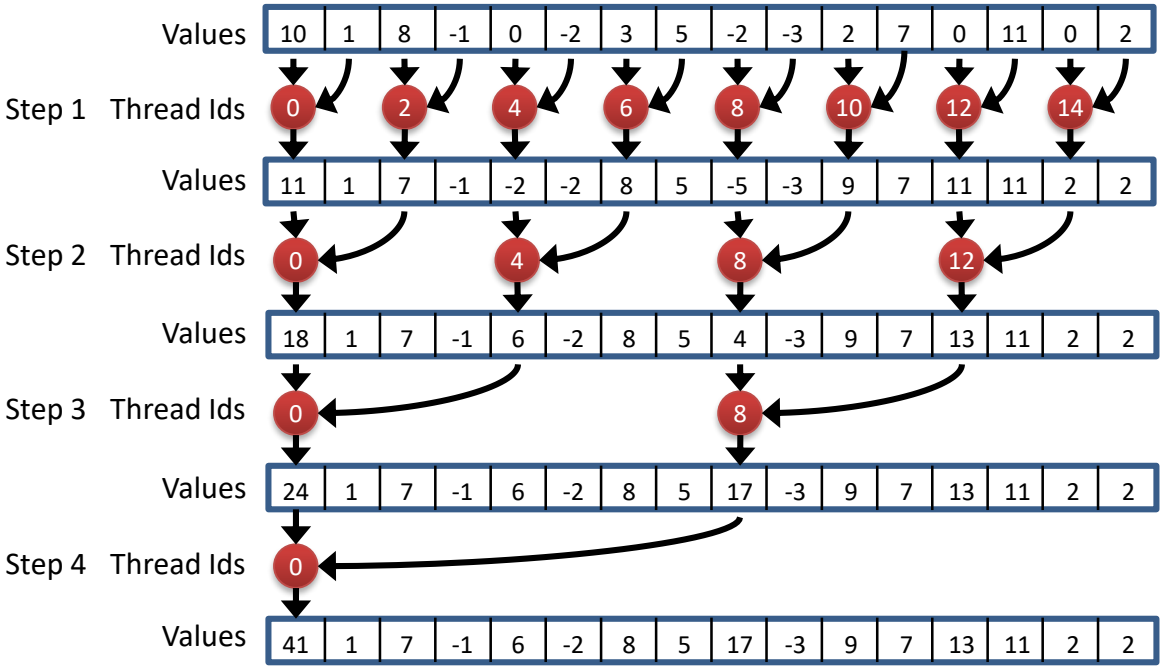
Reduction #2: AtomicShared



Reduction #3: Shared

```
__global__ void reduceShared(const float* input,
          float* result, int elements)
{
  extern __shared__ float data[];
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  data[threadIdx.x] = input[id];
  __syncthreads();

  for(int s = 1; s < blockDim.x; s*=2)
  {
    if(threadIdx.x % (2*s) == 0)
      data[threadIdx.x] += data[threadIdx.x + s];
    __syncthreads();
  }
  if(threadIdx.x == 0)
    atomicAdd(result, x);
}
```

Reduction #3: Shared

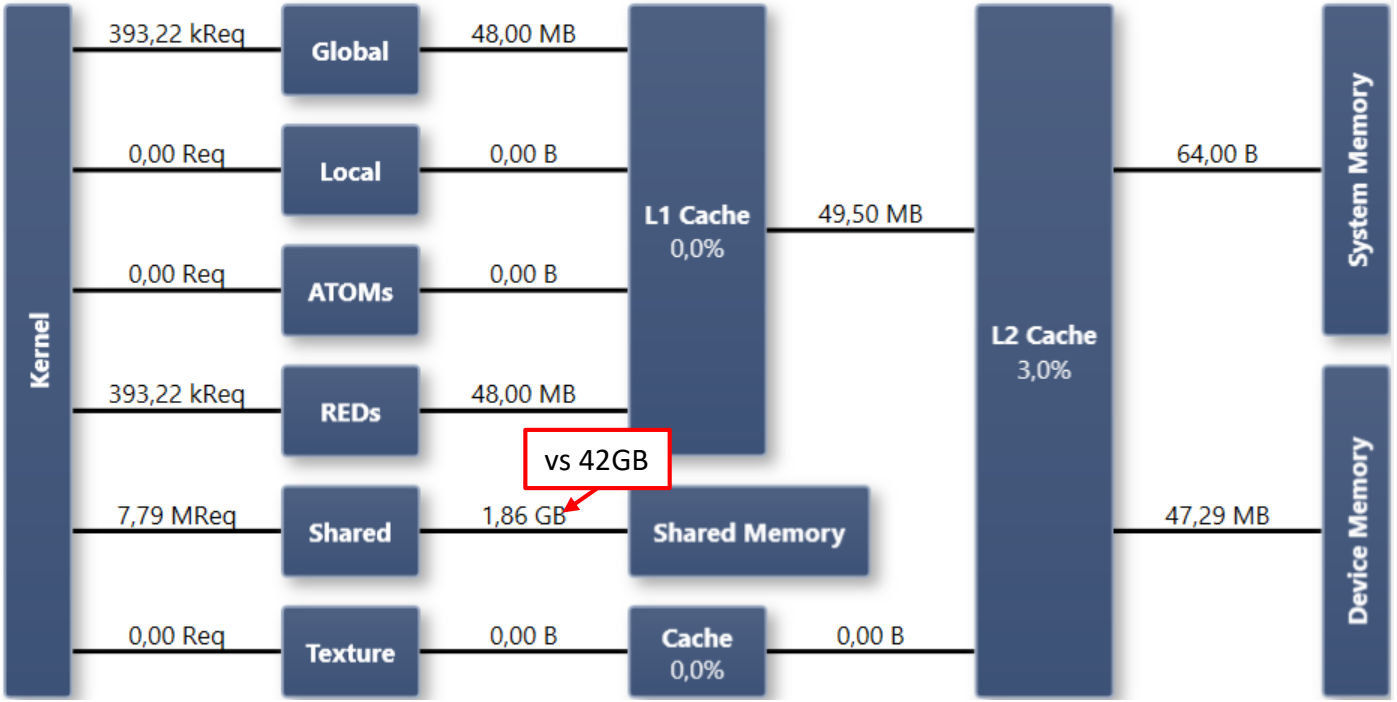


Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000

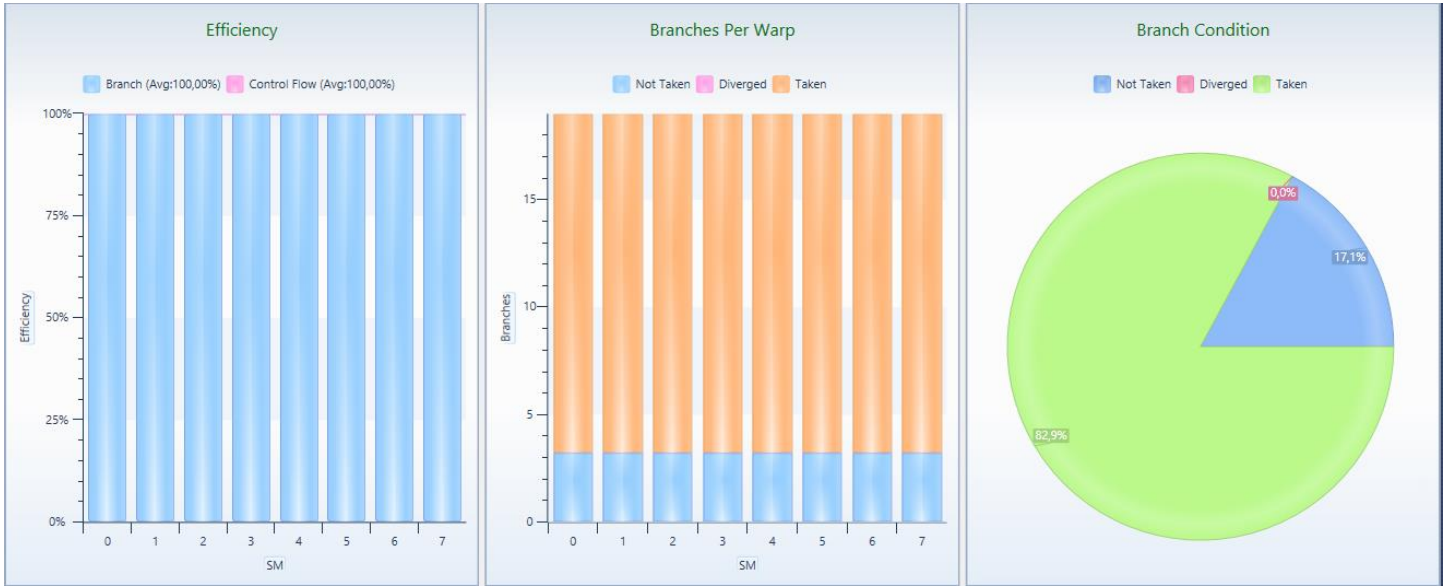
GTX 680 peak: 192GB/S

Reduction #3: Shared



Reduction #3: Shared

- Problem: Divergence

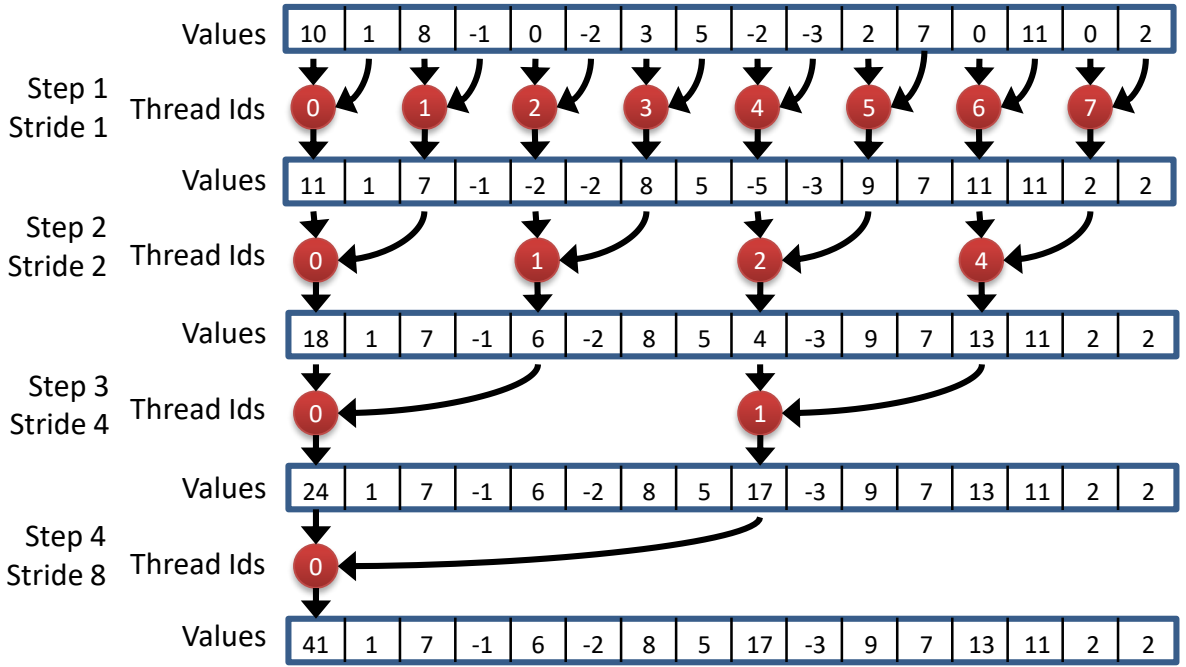


Reduction #4: SharedAntiDivergence

```
for(int s = 1; s < blockDim.x; s*=2)
{
  if(threadIdx.x % (2*s) == 0)
    data[threadIdx.x] += data[threadIdx.x + s];
  __syncthreads();
}
```

```
for(int s = 1; s < blockDim.x; s*=2)
{
  int lid = 2*s*threadIdx.x;
  if(lid < blockDim.x)
    data[lid] += data[lid + s];
  __syncthreads();
}
```

Reduction #4: SharedAntiDivergence

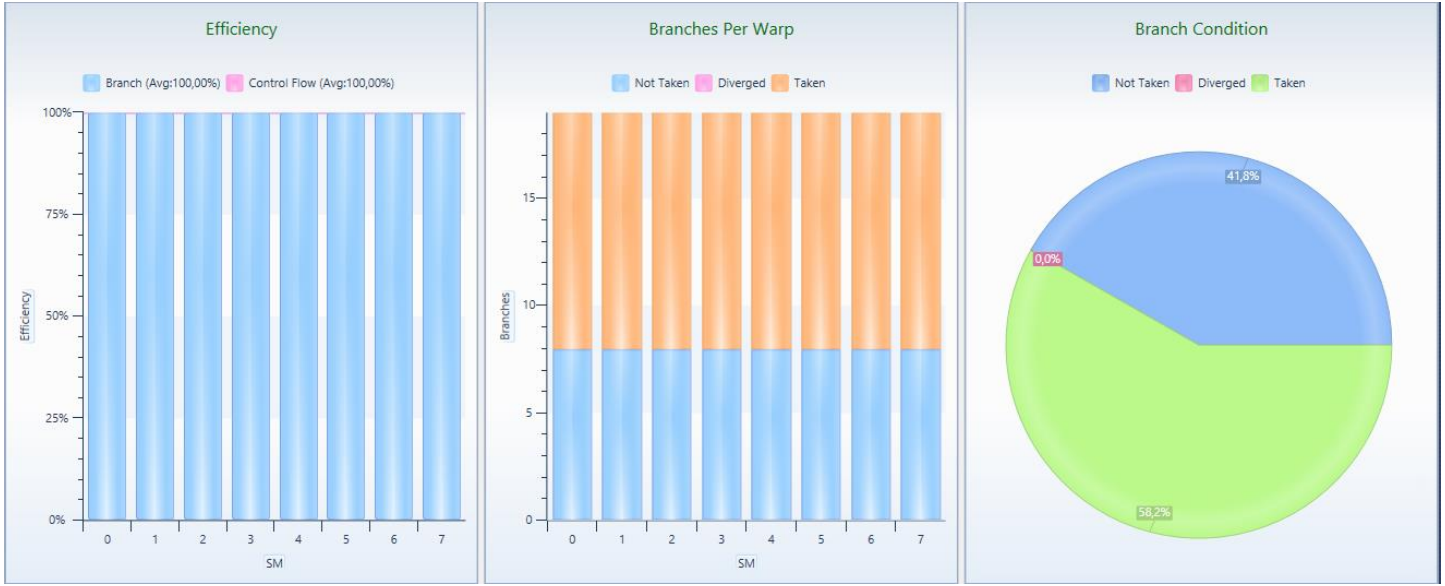


Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235

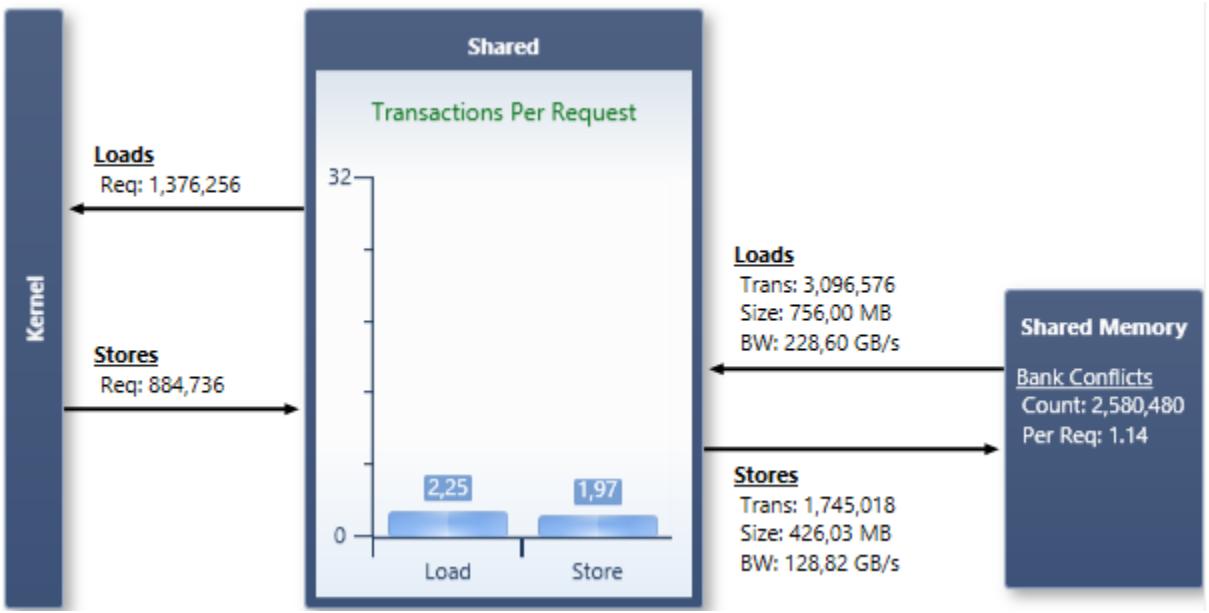
GTX 680 peak: 192GB/S

Reduction #4: SharedAntiDivergence



Reduction #4: SharedAntiDivergence

- Problem: bank conflicts

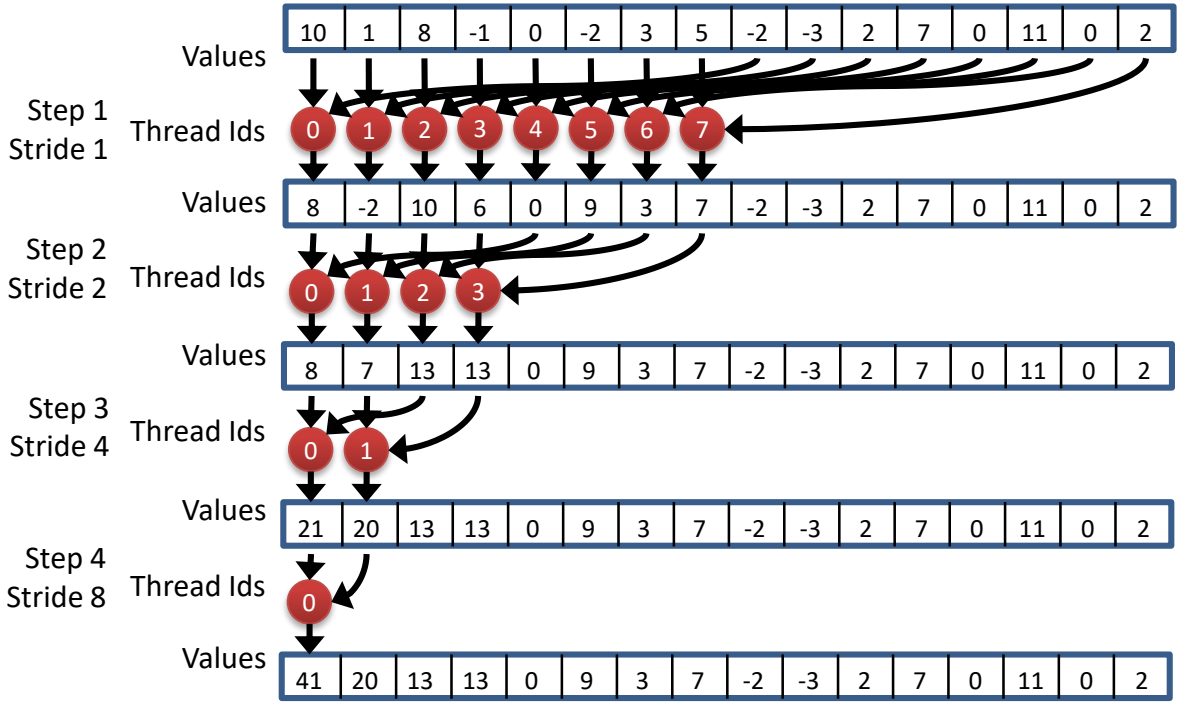


Reduction #5: SharedAntiConflicts

```
for(int s = 1; s < blockDim.x; s*=2)
{
  int lid = 2*s*threadIdx.x;
  if(lid < blockDim.x)
    data[lid] += data[lid + s];
  __syncthreads();
}
```

```
for(int s = blockDim.x/2; s > 0; s/=2)
{
  if(threadIdx.x < s)
    data[threadIdx.x] += data[threadIdx.x + s];
  __syncthreads();
}
```

Reduction #5: SharedAntiConflicts

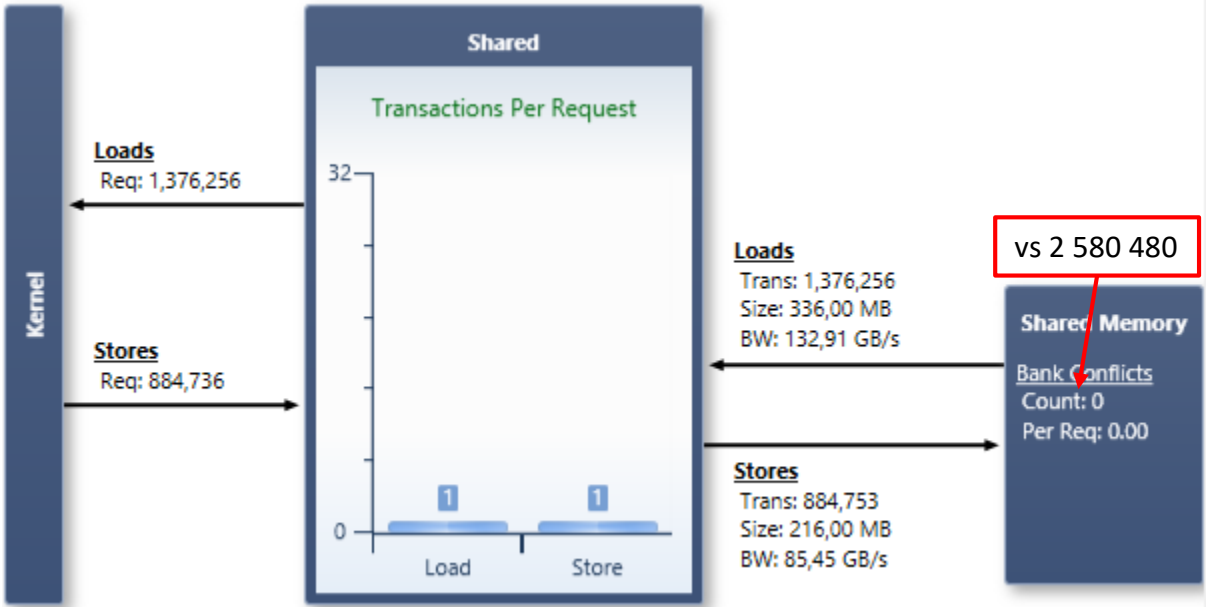


Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618

GTX 680 peak: 192GB/S

Reduction #5: SharedAntiConflicts



Reduction #5: SharedAntiConflicts

- Problem: half of threads idle

```
for(int s = blockDim.x/2; s > 0; s/=2)
{
  if(threadIdx.x < s)
    data[threadIdx.x] += data[threadIdx.x + s];
  __syncthreads();
}
```

Reduction #6: SharedDualLoad

```
__global__ void reduceSharedDualLoad(const float* input,
          float* result, int elements)
{
  extern __shared__ float data[];
  int id = threadIdx.x + blockIdx.x*blockDim.x*2;
  data[threadIdx.x] = input[id] + input[id+blockDim.x];
  __syncthreads();
  for(int s = blockDim.x/2; s > 0; s/=2)
  {
    if(threadIdx.x < s)
      data[threadIdx.x] += data[threadIdx.x + s];
    __syncthreads();
  }
  if(threadIdx.x == 0)
    atomicAdd(result, data[0]);
}
```

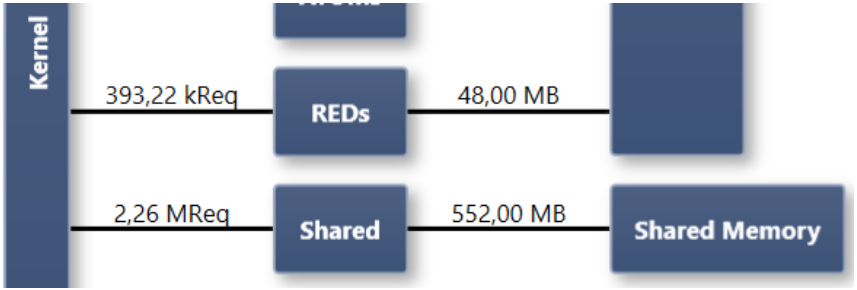
Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618
DualLoad	1.411ms	33.20GB/s	1.949	26.59	3.154

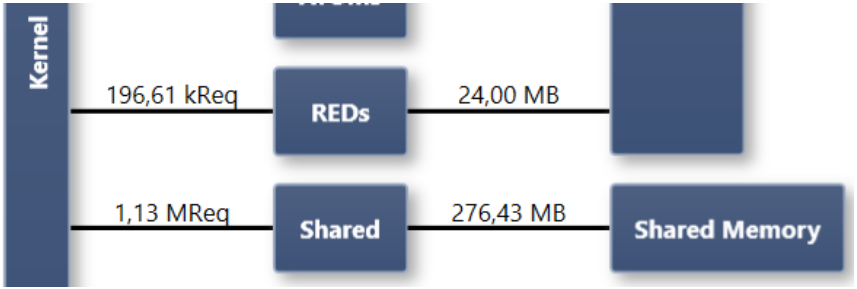
GTX 680 peak: 192GB/S

Reduction #6: SharedDualLoad

SharedAntiConflicts<<<24576,512>>>(…)



SharedDualLoad<<<12288,512>>>(…)



Reduction #6: SharedDualLoad

- 32GB/s vs 192GB/s peak
- What is the bottleneck?
  - Probably instruction overhead
  - Operations that are not loads, stores, or computations
  - Address arithmetic, loops, conditionals
- Idea: unroll loops

Reduction #7: SharedUnrolledLast

```
extern __shared__ volatile float data[];
...
for(int s = blockDim.x/2; s > 32; s/=2)
{
  if(threadIdx.x < s)
    data[threadIdx.x] += data[threadIdx.x + s];
  __syncthreads();
}

if(threadIdx.x < 32)
{
  data[threadIdx.x] += data[threadIdx.x + 32];
  data[threadIdx.x] += data[threadIdx.x + 16];
  data[threadIdx.x] += data[threadIdx.x + 8];
  data[threadIdx.x] += data[threadIdx.x + 4];
  data[threadIdx.x] += data[threadIdx.x + 2];
}
if(threadIdx.x == 0)
  atomicAdd(result, data[0] + data[1]);
```

Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618
DualLoad	1.411ms	33.20GB/s	1.949	26.59	3.154
UnrolledLast	0.978ms	47.90GB/s	1.443	38.35	4.550

GTX 680 peak: 192GB/S

Reduction #7: SharedUnrolledLast

Reduction #8: SharedUnrolledAll (1/2)

```
template<int BlockSize>
__global__ void reduceSharedUnrolledAll(const float* input,
            float* result, int elements)
{
  extern __shared__ volatile float data[];
  int id = threadIdx.x + blockIdx.x*blockDim.x*2;
  data[threadIdx.x] = input[id] + input[id+blockDim.x];
  __syncthreads();

  if(BlockSize >= 512)
  {
    if(threadIdx.x < 256)
      data[threadIdx.x] += data[threadIdx.x + 256];
    __syncthreads();
  }
  if(BlockSize >= 256)
  {
    if(threadIdx.x < 128)
      data[threadIdx.x] += data[threadIdx.x + 128];
    __syncthreads();
  }
```

Reduction #8: SharedUnrolledAll (1/2)

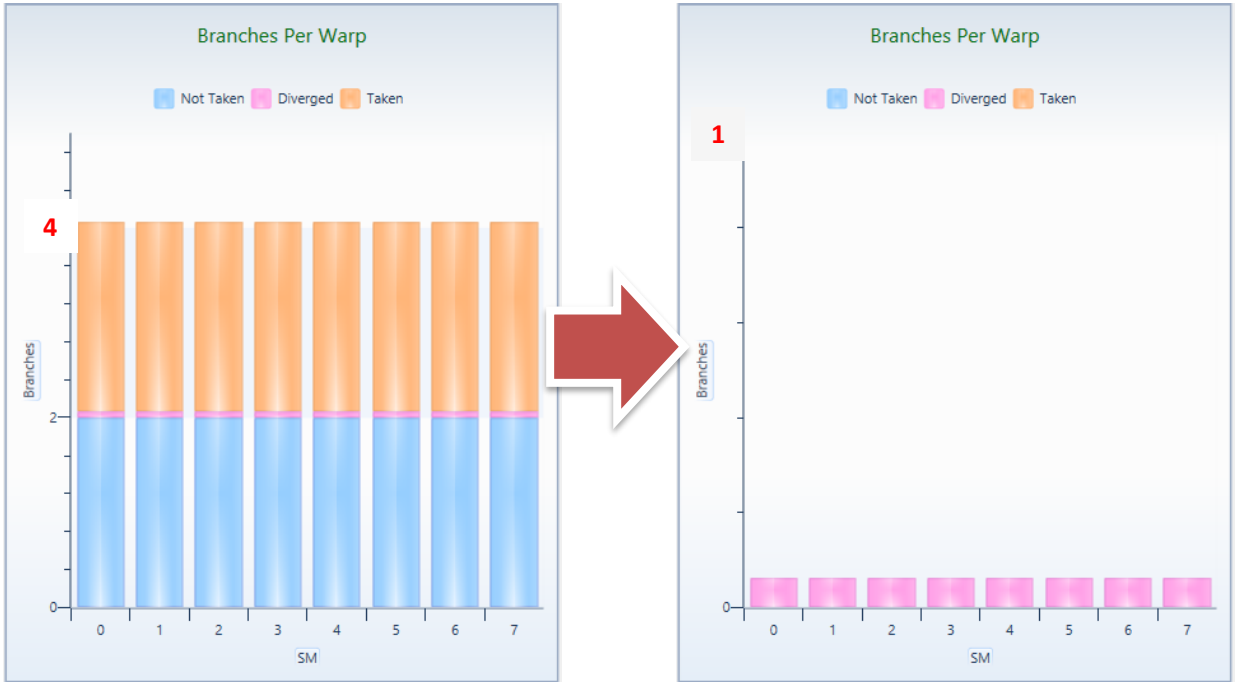
```
  if(BlockSize >= 128)
  {
    if(threadIdx.x < 64)
      data[threadIdx.x] += data[threadIdx.x + 64];
    __syncthreads();
  }

  if(threadIdx.x < 32)
  {
    data[threadIdx.x] += data[threadIdx.x + 32];
    data[threadIdx.x] += data[threadIdx.x + 16];
    data[threadIdx.x] += data[threadIdx.x + 8];
    data[threadIdx.x] += data[threadIdx.x + 4];
    data[threadIdx.x] += data[threadIdx.x + 2];
  }

 if(threadIdx.x == 0)
    atomicAdd(result, data[0] + data[1]);
}
```

Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618
DualLoad	1.411ms	33.20GB/s	1.949	26.59	3.154
UnrolledLast	0.978ms	47.90GB/s	1.443	38.35	4.550
UnrolledAll	0.821ms	57.02GB/s	1.191	45.66	5.417

GTX 680 peak: 192GB/S

Reduction #8: SharedUnrolledAll



Reduction #9: SharedShuffle

```
if(threadIdx.x < 32)
{
  data[threadIdx.x] += data[threadIdx.x + 32];
  data[threadIdx.x] += data[threadIdx.x + 16];
  data[threadIdx.x] += data[threadIdx.x + 8];
  data[threadIdx.x] += data[threadIdx.x + 4];
  data[threadIdx.x] += data[threadIdx.x + 2];
}
```

```
if(threadIdx.x < 32)
{
  d = data[threadIdx.x] + data[threadIdx.x + 32];
  d += __shfl(d, threadIdx.x + 16);
  d += __shfl(d, threadIdx.x + 8);
  d += __shfl(d, threadIdx.x + 4);
  d += __shfl(d, threadIdx.x + 2);
  d += __shfl(d, 1);
}
```

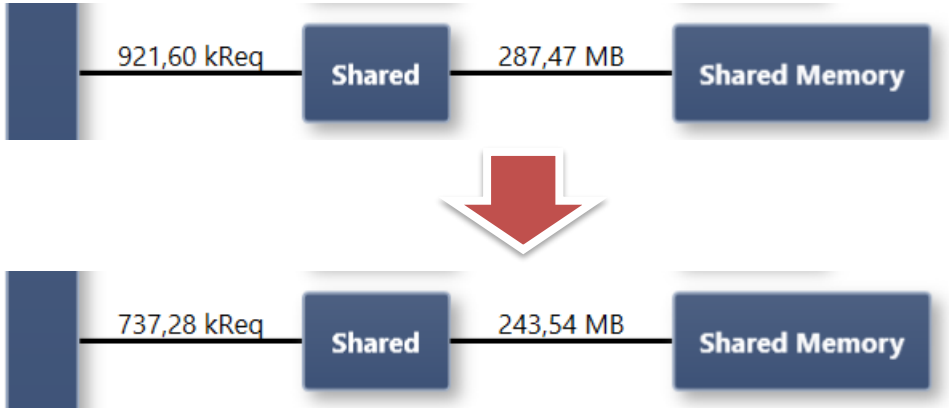
Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618
DualLoad	1.411ms	33.20GB/s	1.949	26.59	3.154
UnrolledLast	0.978ms	47.90GB/s	1.443	38.35	4.550
UnrolledAll	0.821ms	57.02GB/s	1.191	45.66	5.417
Shuffle	0.761ms	61.58GB/s	1.080	49.31	5.850

GTX 680 peak: 192GB/S

Reduction #9: SharedShuffle

- `__shfl(float val, int src)`
  it is possible to directly exchange registers within a warp (CC >= 3.0)



Reduction Complexity

- $\log(N)$ parallel steps
  Each step S does $\frac{N}{2^S}$ independent ops
  → Step complexity is $\mathrm{O}(\log N)$
- $N = 2^D$ operations $= \sum_{S\epsilon[1..D]} 2^{D-S} = N - 1$
  → Work complexity is $\mathrm{O}(N)$ → work efficient
  → Performs as many steps as sequential algorithm
- P threads physically in parallel (P processors)
  → Time complexity is $\mathrm{O}\left(\frac{N}{P} + \log P\right)$
  → For a block $N = P$, *so* $O(\log N)$

Reduction Cost

- Cost of a parallel algorithm is processors × time complexity
  - With N threads
  - → Cost complexity $= N \cdot \mathrm{O}(\log N) = \boldsymbol{O(N \log N)}$
  - → not cost efficient
- Brent's theorem suggests $N/\log N$ threads
  - Each thread does $O(\log N)$ sequential work
  - Then all $N/\log N$ threads cooperate for *O(log N)* steps
  - → Cost $= \mathrm{N}/\log N \cdot O(\log N) + \mathrm{N}/\log N \cdot O(\log N) = \boldsymbol{O(N)}$
  - → Balance between sequential and parallel work
- For limited number of physical processors we see the same effect

Reduction #10: SharedMultiSeq

```
extern __shared__ float data[];
int id = threadIdx.x + blockIdx.x*blockDim.x*2;
data[threadIdx.x] = input[id] + input[id+blockDim.x];
__syncthreads();
```

```
extern __shared__ float data[];
int id = threadIdx.x + blockIdx.x*blockDim.x;
int gridSize = BlockSize*gridDim.x;

float in = 0.0f;
for(; id < elements; id += gridSize)
  in += input[id];

data[threadIdx.x] = in;
__syncthreads();
```

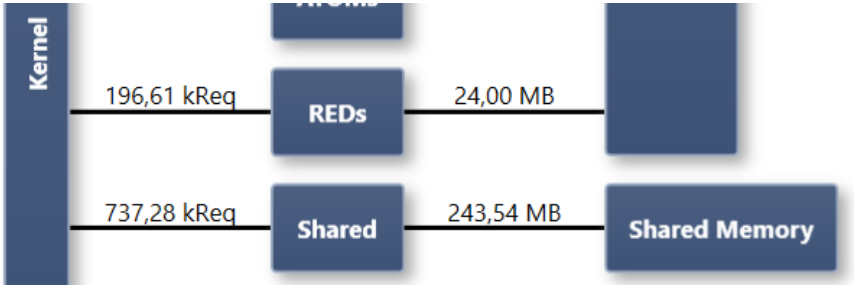
Performance 12M Reduction

	Time ($3*2^{22}$ floats)	Bandwidth	Step speedup	Cumulative Speedup	Speedup to reduction
AtomicGlobal	37.53ms	1.248GB/s	1.000	1.000	
AtomicShared	29.94ms	1.565GB/s	1.254	1.254	
Reduction	4.452ms	10.52GB/s	6.725	8.43	1.000
AntiDivergence	3.604ms	13.00GB/s	1.235	10.41	1.235
AntiConflicts	2.751ms	17.03GB/s	1.310	13.64	1.618
DualLoad	1.411ms	33.20GB/s	1.949	26.59	3.154
UnrolledLast	0.978ms	47.90GB/s	1.443	38.35	4.550
UnrolledAll	0.821ms	57.02GB/s	1.191	45.66	5.417
Shuffle	0.761ms	61.58GB/s	1.080	49.31	5.850
MultiSeq	0.389ms	120.4GB/s	1.956	96.47	11.44

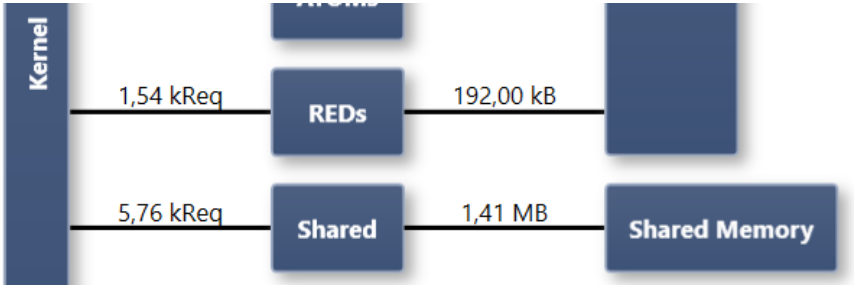
GTX 680 peak: 192GB/S

Reduction #10: SharedMultiSeq

SharedShuffle<<<12288,512>>>(…)



SharedMultiSeq<<<96,512>>>(…)



Questions