CUDA Application Reduction Algorithm

Bedřich Beneš, Ph.D.
Purdue University
Department of Computer Graphics

A Reduction Algorithm

- a reduction algorithm
  extracts a value from an array
- examples: sum, min, max, average…
- Let's assume data is in `int a[n];`

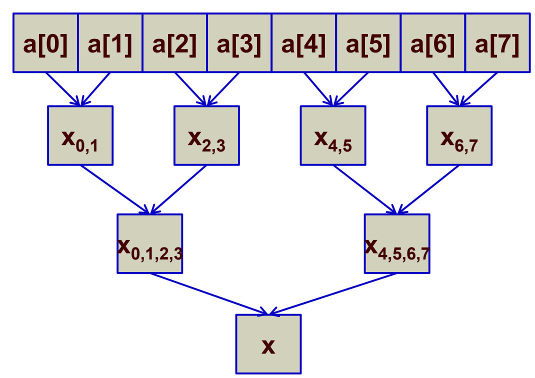
Sequential Reduction Algorithm

A sequential version O(n)

```
x=action(a[0]);
for (int i=1;i<n;i++)
     x=action(a[i]);
//now x stores the result
```


Parallel Reduction Algorithm

A parallel version (soccer tournament) $O(\log_2 n)$



Parallel Reduction Algorithm

Problems:

partial results need to be shared

very low arithmetic intensity

Ideally:

global synchronization

CUDA:

decompose into multiple kernel invocations

Parallel Reduction Algorithm

Metrics of performance:

a) GFLOPs

b) bandwidth

as the arithmetic intensity is low,
bandwidth is a better measure

Parallel Reduction Algorithm

- The operation will be performed per block
- Each block will load the data into shared memory
- All threads work on the block in parallel

Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth

Reduction Algorithm v.1

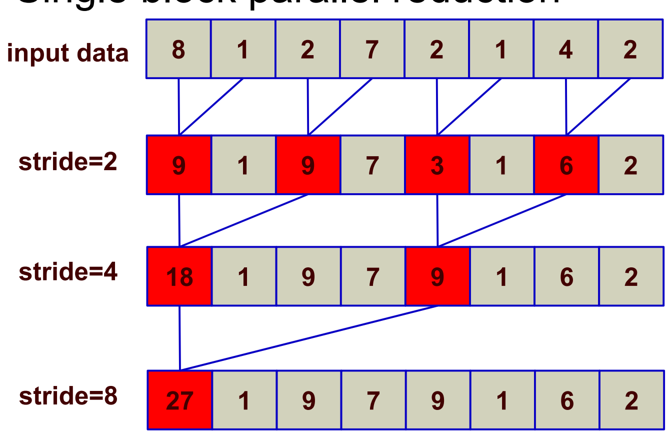
```
//all elements are in the shared memory
unsigned int n=2<<21;//# of integers
size_t size = n*sizeof(int);//datasize [B]
int maxThreads=128; //threads per block
int threads=(n<maxThreads)?n:maxThreads;
int blocks=n/threads; //# of blocks!!
dim3 dimBlock(threads,1,1);
dim3 dimGrid(blocks,1,1);
int smSize=threads*sizeof(int);//shared mem
//mallocs should be here
//note - size of the data is the block size
PR<<<dimGrid,dimBlock,smSize>>>(d);
```


Reduction Algorithm v.1

```
__shared__ int sm[];
__device__ void PR(int *d){
 uint tid=threadIdx.x;
 uint i=blockIdx.x*blockDim.x+threadIdx.x;
 sm[tid]=d[i];//copy to SM
 for (int stride=1;stride<blockDim.x;stride*=2)
 {
  __syncthreads();
  if (t%(2*stride)==0)sm[t]+=sm[t+stride];
 }
 if (tid==0) d[blockIdx.x]=sm[0];//copy back
 //d[blockIdx.x] containts the sum of the block
}
```


Reduction Algorithm v.1

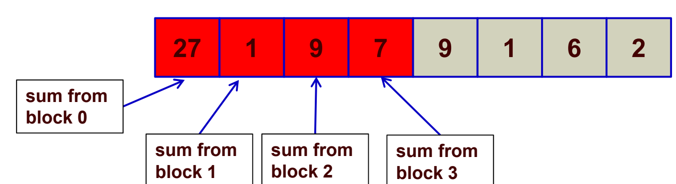
- Single block parallel reduction



Reduction Algorithm v.1

Post process:

- The GPU kernel calculates data per *block*
- Results will be located in the first *block* elements of the global memory
- We will sum them with the same kernel…



Reduction Algorithm v.1

```
if (blocks>1) toDo=1+blocks/128;
              else toDo=0;
for (int i=0;i<toDo;i++){
     threads=
       (blocks<maxThreads)?blocks:maxThreads;
     blocks=blocks/threads;
     dim3 dimBlock(threads,1,1);
     dim3 dimGrid(blocks,1,1);
     PR<<<dimGrid,dimBlock,smemSize>>>(d);
}
cudaMemcpy(hOut,d,s,cudaMemcpyDeviceToHost);
//hOut[0] is the result
```


Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth
Version 1	1	7.800ms	4.28 GB/sec

Reduction Algorithm v.1

Problems:

- thread divergence due to interleaved branch decisions
  `if (t%(2*stride)==0)sm[t]+=sm[t+stride];`
- half of the threads does nothing!
- loop is expensive
- the % operator is slow

Reduction Algorithm v.2

- replace the divergent branch with a non-divergent one

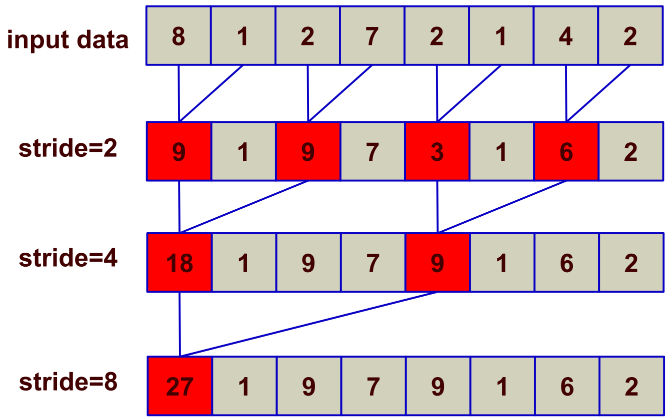
```
if ((tid%(2*stride))==0)
               sm[tid]+=sm[tid+stride];
```

↓

```
int index=2*stride*tid;
if (index<blockDim.x)
               sm[index]+=sm[index+stride];
```

Reduction Algorithm v.2

- Single block parallel reduction



Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth
Version 1	1	7.800ms	4.28 GB/sec
Version 2	1.96	3.975ms	8.41 GB/sec

Reduction Algorithm v.2

Problems:

- ~~thread divergence~~
- half of the threads does nothing!
- the % operator is slow
- shared memory bank conflicts
- loop is expensive

Reduction Algorithm v.3

- replace the strided loop with a reversed one

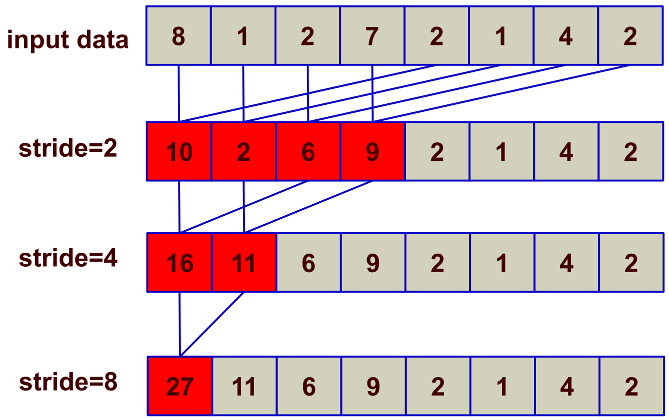
```
for(stride=1;stride<blockDim.x; stride *= 2){
 int ind=2*stride*tid;
 if (ind<blockDim.x) sm[ind]+=sm[ind+stride];
 __syncthreads();
 }
```

↓

```
for (stride=blockDim.x/2;stride>0;stride>>=1){
 if (tid<stride) sm[tid]+=sm[tid+stride];
 __syncthreads();
}
```

Reduction Algorithm v.3

- Single block parallel reduction



Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth
Version 1	1	7.800 ms	4.28 GB/sec
Version 2	1.96	3.975 ms	8.41 GB/sec
Version 3	2.94	2.650 ms	12.43 GB/sec

Reduction Algorithm v.3

Problems:

- ~~thread divergence~~
- half of the threads does nothing!
- ~~the % operator is slow~~
- ~~sm bank conflicts~~
- loop is expensive

Reduction Algorithm v.4

let's make busy all threads in the first step

1) use only the half the blocks and

2) do the first reduction during
the load from GM
(replace single load with two loads)

Reduction Algorithm v.4

```
uint tid=threadIdx.x;
uint i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
sm[tid]= d[i]+d[i+blockDim.x];
__syncthreads();
for (stride=blockDim.x/2;stride>0;stride>>=1){
     if (tid<stride) sm[tid]+=sm[tid+stride];
      __syncthreads();
}
// write result for this block to global mem
if (tid == 0) d[blockIdx.x]=sm[0];
```


Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth
Version 1	1	7.800 ms	4.28 GB/sec
Version 2	1.96	3.975 ms	8.41 GB/sec
Version 3	2.94	2.650 ms	12.43 GB/sec
Version 4	5.55	1.405 ms	23.90 GB/sec

Reduction Algorithm v.4

- Problems
- It has low arithmetic intensity, so the bandwidth should be better
- But 23.9 GB/sec is about 24% of peak performance

Reduction Algorithm v.5

- Let’s unroll the loops!
- Number of active threads decreases with the number of iterations
- for stride<=32 we have only one warp
- warp runs the same instruction (SIMD)
- for stride<=32 __syncthreads() is not necessary
- Let’s unroll last 6 iterations

Reduction Algorithm v.5

```
uint tid=threadIdx.x;
uint i=blockIdx.x*(blockDim.x*2)+threadIdx.x;
sm[tid] = d[i]+d[i+blockDim.x];
__syncthreads();
for (stride=blockDim.x/2;stride>32;stride>>=1)
{
  if (tid<stride) sm[tid]+=sm[tid+stride];
   __syncthreads();
}
```


Reduction Algorithm v.5

```
if (tid < 32)
{
     sm[tid]+=sm[tid+32];
     sm[tid]+=sm[tid+16];
     sm[tid]+=sm[tid+8];
     sm[tid]+=sm[tid+4];
     sm[tid]+=sm[tid+2];
     sm[tid]+=sm[tid+1];
}
// write result for this block to global mem
if (tid == 0) d[blockIdx.x]=sm[0];
}
```


Comparison

- 32 MB of data (2^{22} integers)
- Tesla C1060
- Data block size 128
- Peak bandwidth 102 GB/sec

version \ values	Speedup	Time	Bandwidth
Version 1	1	7.800 ms	4.28 GB/sec
Version 2	1.96	3.975 ms	8.41 GB/sec
Version 3	2.94	2.650 ms	12.43 GB/sec
Version 4	5.55	1.405 ms	23.90 GB/sec
Version 5	5.89	1.325	24.33 GB/sec

Additional measurements

- 128 MB of data (2^{24} integers)
- Tesla C1060
- Data block size 512
- Bandwidth 50 GB/sec
- That is 50% of peak bandwidth 102 GB/sec

Conclusions

- Optimizations lead to a great speedup
- Algorithmic optimizations (addressing, cascading, etc) ~ 3x speedup
- Loop unrolling another 2x speedup
- First optimize, then unroll the loops!

Reading

- Harris, M., *Optimizing Parallel Reduction in CUDA* (NVIDIA Dev. technology)
- Kirk, D.B., Hwu, W.W., *Programming Massively Parallel Processors,* NVIDIA, Morgan Kaufmann 2010