GPU Computing

Patterns for massively parallel programming (part 2)

Reduction Pattern

E. Carlinet, J. Chazalon {`firstname.lastname@lrde.epita.fr`}

April 22

EPITA Research & Development Laboratory (LRDE)

Slides generated on April 20, 2022

Reference

Reference

We present (and explain) here a subset of the optimizations described in the presentation “Optimizing Parallel Reduction in CUDA”, from Mark Harris, NVIDIA.

Intuition for reduction pattern

What is a reduction?

Reduction combines every element in a collection into one element using an associative operator.

Sequential computation of a global sum

$$s = (\dots((B[0] + B[1]) + B[2]) + \cdots + B[n])$$

```
sum = 0;
for (int i = 0; i < n; ++i)
    sum += B[i];
```

Reduction Pattern: solution 1

```
__global__ void reduce(const int* buffer, int size, int* sum)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  *sum += buffer[x];
}
```

Is this correct?

Reduction Pattern: solution 1

```
__global__ void reduce(const int* buffer, int size, int* sum)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  *sum += buffer[x];
}
```

Is this correct?

No: there is a data race

(and a missing boundary check, but this is not the point here)

Data Race

Race condition A computational hazard that arises when the results of the program depend on the timing of uncontrollable events, such as the execution order of threads.

A **data race** is a particular case of **race condition** which depends on data.

Thread 1

1. read `tmp` $\leftarrow$ `*sum`
2. compute `tmp = tmp + buffer[x]`
3. write `tmp` $\rightarrow$ `*sum`

Thread 2

1. read `tmp` $\leftarrow$ `*sum`
2. compute `tmp = tmp + buffer[x]`
3. write `tmp` $\rightarrow$ `*sum`

We need to ensure that each of those read-compute-write sequences are **atomics** to avoid **data races**.

Atomics reminder

Atomics

- Read, modify, write in one operation
- Cannot be mixed with accesses from other thread
- On global memory, and shared memory
- Atomic operations to the same address are serialized

Operations

Arithmetic	atomic{Add,Sub,Inc,Dec}
Min-max	atomic{Min,Max}
Primitives	atomic{Exch,CAS}
Bitwise	atomic{And,Or,Xor}

Reduction Pattern Corrected

Accumulation in global memory

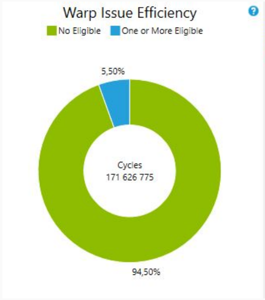
```
__global__ void reduce0(const int* buffer, int size, int* sum)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (x < size) atomicAdd(sum, buffer[x]);
}
```

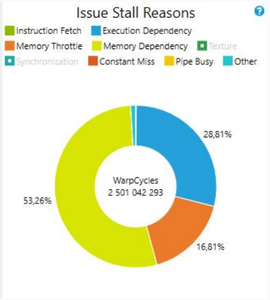
Reduction Pattern Corrected

Accumulation in global memory

```
__global__ void reduce0(const int* buffer, int size, int* sum)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (x < size) atomicAdd(sum, buffer[x]);
}
```

Analysis *(old Maxwell benchmark)*





Time: **5.619** ms

Correct result but

high contention on the global atomic variable

→ the execution is actually **sequential**!

Global atomics: Is this really parallel?

This version will produce the right result.

However, **is it really parallel?**

How our global atomic instruction is executed:

1. lock memory “cell”,
2. read old value (approx. 100 cycles from L2 cache, 1000 from DRAM),
3. compute new value,
4. write new value (approx. 100 cycles from L2 cache, 1000 from DRAM)
5. release the memory “cell”

Memory “cell” =? cache line (“segments”) = 128 bytes = 32 `uint32`

Our kernel generates a lot of collisions on global memory.

This forces the serialization of much of the work!

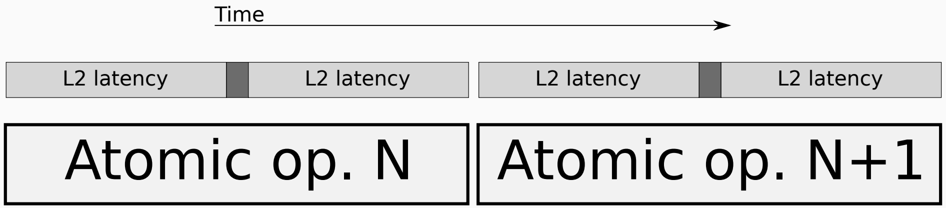


Figure 1: Timing of atomic operation using L2 cache

Output Privatization

Leverage Shared Memory

Atomic operations are **much much faster on shared memory**.

However, shared memory is only shared within a thread block:
this requires to adapt the code.

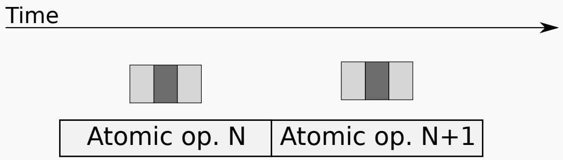
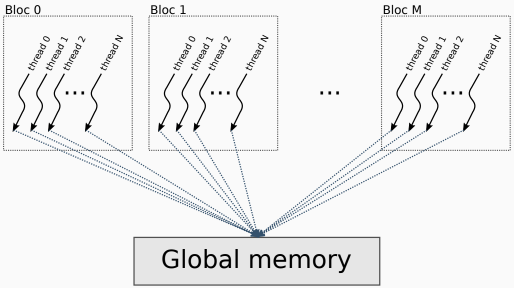
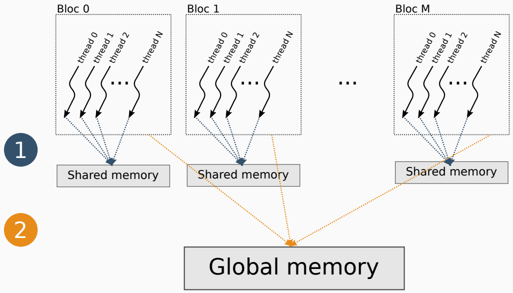


Figure 2: Timing of atomic operation using shared memory

Motivation for output privatization





Instead of heavy contention and serialization because of atomics to global memory...

...use local memory to:

1. accelerate atomics
2. reduce contention and serialization of writes to global memory

Intermede How to allocate shared memory?

Code for static allocation

```
__global__ void histo(int* buf, int w, int h, int pitch, int max_iter, int* hist)
{
   __shared__ int localHist[MAX_ITER+1]; // compile-time constant
}
```

Intermede How to allocate shared memory?

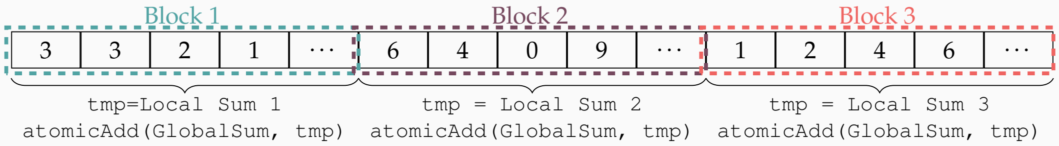
Code for static allocation

```
__global__ void histo(int* buf, int w, int h, int pitch, int max_iter, int* hist)
{
   __shared__ int localHist[MAX_ITER+1]; // compile-time constant
}
```

Code for dynamic allocation

```
__global__ void histo(int* buf, int w, int h, int pitch, int max_iter, int* hist)
{
   extern __shared__ int localHist[]; // no allocation
}

int main() {
  // ...
  histo<<<nBlocks, threadsPerBlock, (max_iter + 1) * sizeof(int)>>>(/* ... */);
  // dynamic allocation of shared memory (3rd <<<...>>> parameter)
}
```

Reduction Pattern V2: Output privatization

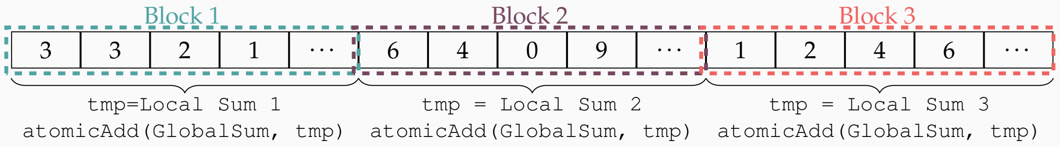


Sum locally on each block and sum at the end on the global variable

```
void reduce1(const int* buffer, int size, int* sum) {
  __shared__ int local_sum;
  // Initialization of local/shared memory
  if (threadIdx.x == 0) local_sum = 0;

  // Local accumulation
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (x < size) atomicAdd(&local_sum, buffer[x]);

  // Commit to global memory
  if (threadIdx.x == 0) atomicAdd(sum, local_sum);
}
```

. . . Is this correct?

Reduction Pattern V2: Output privatization



Sum locally on each block and sum at the end on the global variable

```
void reduce1(const int* buffer, int size, int* sum) {
  __shared__ int local_sum;
  // Initialization of local/shared memory
  if (threadIdx.x == 0) local_sum = 0;

  // Local accumulation
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (x < size) atomicAdd(&local_sum, buffer[x]);

  // Commit to global memory
  if (threadIdx.x == 0) atomicAdd(sum, local_sum);
}
```

. . . Is this correct?

No! we need synchronization

Synchronization reminder

Execution synchronization

`__syncthreads()` forces all threads in a block to reach a given code line before they continue.

This synchronization mechanism also forces *memory synchronization*.

Synchronization reminder

Execution synchronization

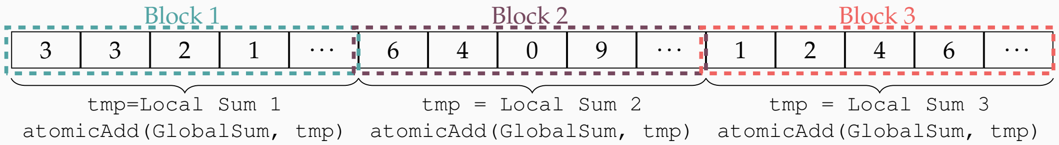
`__syncthreads()` forces all threads in a block to reach a given code line before they continue.

This synchronization mechanism also forces *memory synchronization*.

Memory synchronization

- Nvidia GPUs (and compiler) implement a relaxed consistency model
- No global ordering between memory accesses
- Threads may not see the writes/atomics in the same order
- Enforcing memory ordering:
    - `__threadfence_block`
    - `__threadfence`
    - `__threadfence_system`
- Make writes preceding the fence appear before writes following the fence for the other threads at the block / device / system level
- Make reads preceding the fence happen after reads following the fence

Reduction Pattern V2: Output privatization (with syncs)



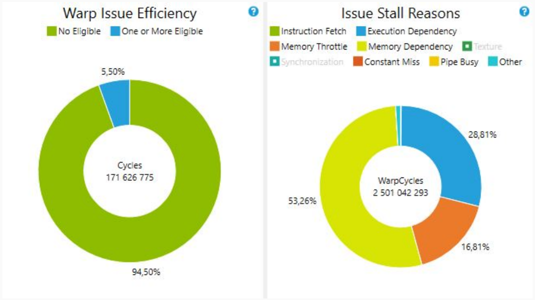
- Accumulate locally on each block
- Accumulate at the end on the global variable

```
void reduce1(const int* buffer, int size, int* sum)
{
  __shared__ int local_sum;
  if (threadIdx.x == 0) local_sum = 0;
  __syncthreads(); // <<<<<

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (x < size) atomicAdd(&local_sum, buffer[x]);

  __syncthreads(); // <<<<<
  if (threadIdx.x == 0) atomicAdd(sum, local_sum);
}
```

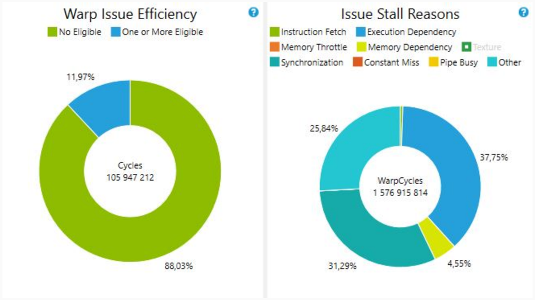
Correct!

Reduction Pattern V2: Output privatization

Global atomic



Output privatization



Method	Time
Global atomics	5.619 ms
Local atomics	3.465 ms

Less collisions, but local summations remain sequential!

Reduction Trees

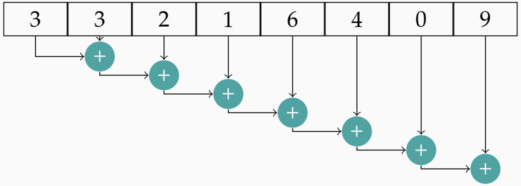
Reduction functions and trees

We need to think a bit more to really achieve a parallel computation.

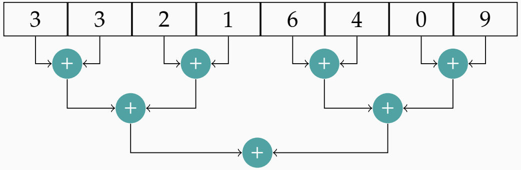
We can leverage **properties of reduction functions** to implement different computation flows:

- reduction functions are functions like **min**, **max**, **sum** or **product**
- they are **associative** and **commutative**
- they have a well-defined **identity value** (e.g., 0 for sum)

Sequential reduction



Parallel reduction using a tree



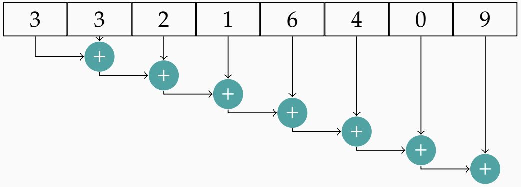
It's all about moving parenthesis!

Sequential: $(\dots((0 + A[0]) + A[1]) + \dots + A[n])$

Tree: $(\dots((A[0] + A[1]) + \dots) + \dots + (\dots + (A[n-1] + A[n]))\dots)$

Complexity in steps and operations

Sequential reduction

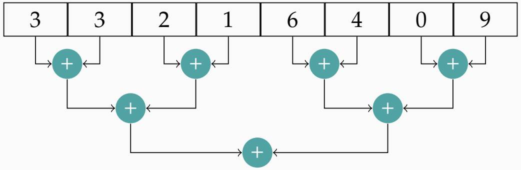


Number of operations: $N-1$

Number of steps: $N-1$

Number of threads required: 1

Parallel reduction using a tree



Number of operations: $N-1$

Number of steps: $log_2(N)$

Number of threads required (peak): $N/2$

The tree parallel version is:

- work efficient: same number of operations as the efficient sequential version
- not resource efficient:
  average number of thread $((N-1)/log_2(N)) \ll$ peak requirement $(N/2)$

Proof of number of operations 🌶️

$$\begin{aligned}\text{Num. op} &= \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + 1 \\ &= \frac{N}{2}\cdot(\frac{1}{2})^0 + \frac{N}{2}\cdot(\frac{1}{2})^1 + \frac{N}{2}\cdot(\frac{1}{2})^2 + \dots + \frac{N}{2}\cdot(\frac{1}{2})^{k-1}\end{aligned}$$

with $k = log_2(N)$.

Knowing that:

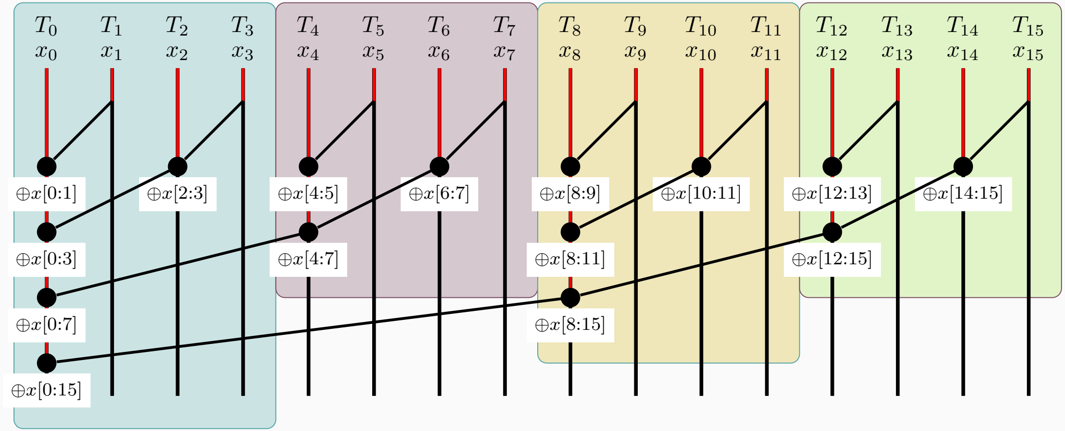
$$\sum_{i=0}^{n-1} ar^i = a\frac{1-r^n}{1-r}$$

we have $a = \frac{N}{2}$ and $r = \frac{1}{2}$.

Hence:

$$\begin{aligned}\text{Num. op} &= \frac{N}{2}\cdot(\frac{1}{2})^0 + \frac{N}{2}\cdot(\frac{1}{2})^1 + \frac{N}{2}\cdot(\frac{1}{2})^2 + \dots + \frac{N}{2}\cdot(\frac{1}{2})^{k-1} \\ &= \frac{N}{2}(\frac{1-(\frac{1}{2})^k}{1-\frac{1}{2}}) \\ &= N(1-\frac{1}{2^k}) \\ &= N(1-\frac{1}{N}) \\ &= N-1 \quad \square\end{aligned}$$

Reduction Pattern: tree reduction without atomics



- Use a local sum without atomics
    - Map a reduction tree to compute units (threads)
- Add to a global atomic once for each block

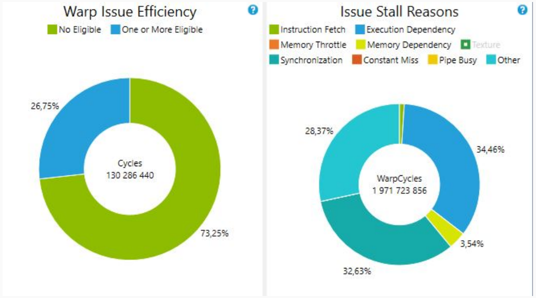
Reduction Pattern: tree reduction without atomics

```
__global__ void reduce2(const int* buffer, int size, int* sum)
{
    extern __shared__ int partialSum[]; // len=blockDim.x

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    partialSum[tid] = buffer[i];
    __syncthreads();

    // Collaborative reduction
    ...
}
```

Reduction Pattern: tree reduction without atomics

```
__global__ void reduce2(const int* buffer, int size, int* sum)
{
    extern __shared__ int partialSum[]; // len=blockDim.x

    // each thread loads one element from global to shared mem
    ...

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

    // write result for this block to global mem
    if (tid == 0) atomicAdd(sum, partialSum[0]);
}
```

Reduction Pattern: tree reduction without atomics



Method	Time
Global atomic	5.619 ms
Local atomics	3.465 ms
Local No Atomic	4.262 ms

What is happening?

The (naive) tree version is slower than the locally sequential version!

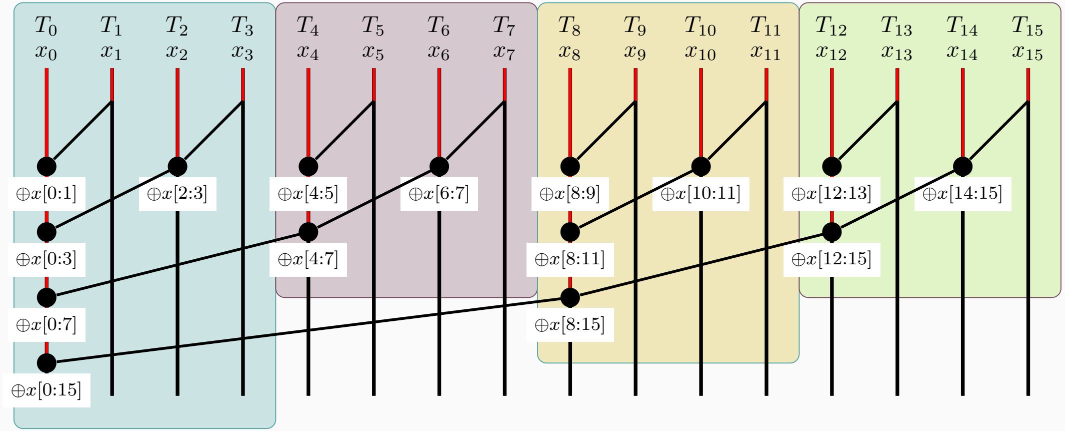
Any idea?

SP starvation

In each iteration, two control flow paths will be sequentially traversed for each warp

- Threads that perform addition and threads that do not
- Threads that do not perform addition **still consume execution resources**

Active threads



T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13} T_{14} T_{15}
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
$\oplus x[0{:}1]$ $\oplus x[2{:}3]$ $\oplus x[4{:}5]$ $\oplus x[6{:}7]$ $\oplus x[8{:}9]$ $\oplus x[10{:}11]$ $\oplus x[12{:}13]$ $\oplus x[14{:}15]$
$\oplus x[0{:}3]$ $\oplus x[4{:}7]$ $\oplus x[8{:}11]$ $\oplus x[12{:}15]$
$\oplus x[0{:}7]$ $\oplus x[8{:}15]$
$\oplus x[0{:}15]$

SP starvation

Half or fewer of threads will be executing after the first step

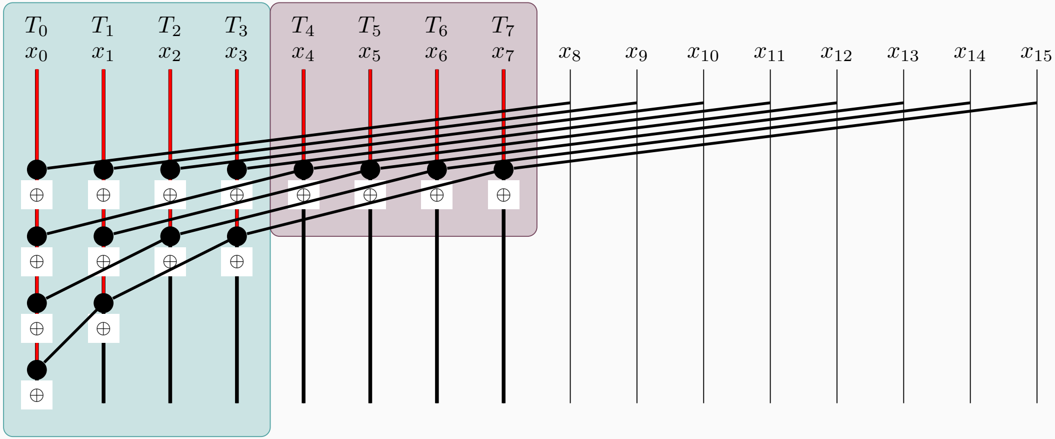
- All odd-index threads are disabled after first step
- When typically used on buffers of 2048 items, after the 5th step, entire warps in each block will fail the `if` test → poor resource utilization but no divergence
- This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

With only a few threads active in each warp, **our Streaming Processors are starving!**

- **We need to group active threads in the same warps.**
- This will enable us to stop useless warps as soon as possible.

Efficient Reduction Trees

Resource efficient version



```
__global__ void reduce3(const int* buffer, int size, int* sum)
{
    extern __shared__ int partialSum[]; // len=blockDim.x

    // Collaborative loading
    // (unchanged)

    // Collaborative reduction
    for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2)
    {
        if (tid < stride)
            partialSum[tid] += partialSum[tid + stride];
        __syncthreads();
    }

    // Write to global memory
    // (unchanged)
}
```

A Quick Analysis

For a 1024 thread block

- No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
- The final 5 steps will still have divergence
    - Can use warp-level optimization then (warp shuffle)
    - Requires warp synchronization with CC 7.0+

Performance analysis:

Method	Time
Global atomic	5.619 ms
Local atomics	3.465 ms
Local No Atomic	4.262 ms
Local No Atomic Grouped	MISSING

Missing on old benchmark, better than reduce2, but wait until the end...

Avoiding idle threads

Optimize the first loop

```
__global__ void reduce3(const int* buffer, int size, int* sum)
{
   ...
    for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2)
    {
        if (tid < stride)
    ...
}
```

Half of the threads are idle in the first loop iteration.

Let’s load twice!

Double loading

```
__global__ void reduce4(const int* buffer, int size, int* sum)
{
    extern __shared__ int partialSum[]; // len=blockDim.x * 2

    // Collaborative loading
    int t = threadIdx.x;
    int start = 2 * blockIdx.x * blockDim.x;
    int x = start + t;
    partialSum[t] = (x < size) ? buffer[x] : 0;
    x += blockDim.x;
    partialSum[t + blockDim.x] = (x < size) ? buffer[x] : 0;
    __syncthreads();

    // Collaborative reduction
    for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2)
    // (unchanged)
}
```

Requires to half the number of blocks!

But wait, this is stupid...

This idea has 2 advantages:

1. it reduces the number of blocks (less scheduling overhead)
2. it avoid idle threads in the first iteration of the loop

but

- it puts more pressure on shared memory
- it just postpones the problem

Algorithm Cascading

Limit global collision

What happens with very large input arrays?

Lot of global atomics.

How to avoid this?

Global array, one cell for each block

- No more locks
- But requires a second level of reduction AND requires extra writes

More work per thread

- Just fire enough blocks to hide latency
- Sequential reduction, then tree reduction
- “algorithm cascading”

Algorithm Cascading

Perform first reduction during the collaborative loading.

Warning: kernel launch parameters must be scaled accordingly!

`N <= blockDim.x * gridDim.x * workPerThread`

```
__global__ void reduce5(const int* buffer, int size, int* sum) {
    extern __shared__ int partialSum[]; // len=blockDim.x

    // Cascading
    int t = threadIdx.x;
    int x = blockIdx.x * blockDim.x + t;
    int gridSize = blockDim.x * gridDim.x;
    partialSum[t] = 0;  // private to thread: no need to sync!
    while (x < size) { // <<<< auto adjust work per thread!
      partialSum[t] += buffer[x];
      x += gridSize;
    }
    __syncthreads();
    //...
}
```

Turning a reduction into (mostly) a map

This cascading strategy scales beautifully: we just need to fire enough blocks to hide the latency of data loading.

All threads are active, and perform one operation per data item, except at the end of the buffer.

Some extra optimizations are still possible...

Extra optimizations

Loop unrolling 🌶️

Fight for ancillary instructions overhead: address arithmetic and loop overhead that are not core computation (like loads, stores or arithmetics)

Solution: loop unrolling

- Unroll tree reduction loop for the last warp (less synchronization needed)
- Unroll all tree reduction loops (need to know block size)
- Unroll the sequential reduction loop (knowing the work per thread)

Loop unrolling 🌶️

```
__global__ void reduce6(const int* buffer, int size, int* sum) {
{
    extern __shared__ int partialSum[];

    // Loading + linear reduction
    ...

    for (unsigned int s = blockDim.x/2; s > 32; s >>=1)
    {
        if (tid < s)
            partialSum[tid] += partialSum[tid+s];
        __syncthreads();
    }
    if (tid < 32) warpReduce4(partialSum, tid);

    ...
}
```

```
__device__ void warpReduce(volatile int* partialSum, int tid)
{
    partialSum[tid] += partialSum[tid + 32];
    partialSum[tid] += partialSum[tid + 16];
    partialSum[tid] += partialSum[tid + 8];
    partialSum[tid] += partialSum[tid + 4];
    partialSum[tid] += partialSum[tid + 2];
    partialSum[tid] += partialSum[tid + 1];
}
```

And there are even more warp tricks available.

Summary

Reduction summary

It is easy to be wrong!

Algorithmic optimizations:

- coalesced accesses
- output privatization
- group active threads
- algorithm cascading

Code optimizations:

- Loop unrolling

Benchmark on target hardware

- minimize compute time for compute-bound programs
- maximize bandwidth for memory-bound programs

Performance summary 🌶️🌶️

Time to compute the sum of 132 M integers. GTX 1060, Compute Capabilities 7.5

Variant	function	time
Global atomicAdd	reduce0	7.02 ms
Output privatization	reduce1	7.68 ms
Tree red., thread divergence	reduce2	29.45 ms
Tree red., NO thread div.	reduce3	17.69 ms
Tree red., NO thread div., dlb load	reduce4	x
Tree red., NO thread div., add twice	not shown	9.32 ms
Tree red., NO thread div., cascading	reduce5	4.65 ms
Tree red., NO thread div., add twice, warp unroll	≈reduce6	6.58 ms
Tree red., NO thread div., cascading, full unroll	not shown	4.55 ms
Cascading and global atomic add	reduce7	4.60 ms
Cascading, warp tree red., avoid idle warps	reduce8	0.56 ms

What?

Global are cheaper than large sync nowadays 🌶️🌶️

```
__global__ void reduce13(int* sum, int* buffer, int size)
{
    int private_acc = 0;

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + tid;
    unsigned int gridSize = blockDim.x*gridDim.x;

    // Cascading
    while (i < size){
        private_acc += buffer[i];
        i+= gridSize;
    }

    atomicAdd(sum, private_acc);
}
```

Avoiding idle warps 🌶️🌶️🌶️

```
__global__ void reduce14(int* sum, int* buffer, int size)
{  // Launch with 32 threads per block, ceil(size / 1024) blocks
    __shared__ int block_sum;
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    if (tid == 0) block_sum = 0;
    __syncthreads();

    int thread_sum = 0; // Cascading and auto compiler unrolling
    for (int k = 0; k < 32; ++k)
        thread_sum += buffer[i + k * 32];

    atomicAdd(&block_sum, thread_sum); // Block

    __syncthreads();
    if (tid == 0) atomicAdd(sum, block_sum);
}
```

Take home messages

- Cascading is great — well map is great
- Don’t optimize too soon — benchmarking is more important — architecture changes quickly
- Pick the right tree-based reduction algorithm
- Context switching is cheap
- Less (arithmetic) operations = less work = faster kernels (obvious, isn’t it?)