CUDA Memory Model

Or how “our strength grows out of our weakness”

by Georgy Evtushenko @g_evtushenko
on January 5, 2021

» Motivation

Performance A significant slowdown is caused by excess memory ordering

Correctness One source of non-deterministic concurrency bugs is a misunderstanding of the architecture's shared memory consistency model i.e. what values can be read from shared memory when issued concurrently with other reads and writes.[1]

[1] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. 2011. ISBN: 1608455645.

» Definitions

Read operation	All variants of ld instruction and atom instruction.
Write operation	All variants of st instruction and atom instruction.
Memory operation	A read or write operation.
Memory model	Set of rules defining how memory operations on shared memory from multiple threads are processed. Tradeoff between ease of programmability and efficiency.
Memory order	Total order of memory operations.

» Sequential consistency

"… the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."[2]

Thread 1

```
x = 1
r1 = y
```

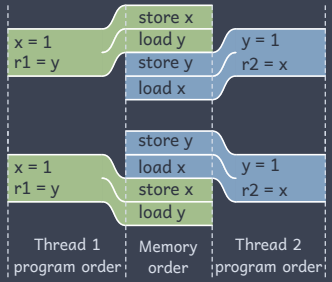
Thread 2

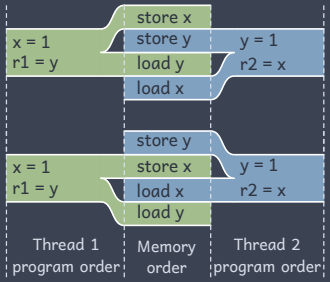
```
y = 1
r2 = x
```

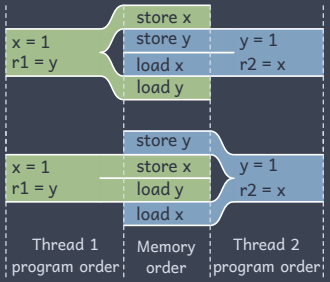
[2] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28 (1979), pp. 690–691.

» Sequential consistency

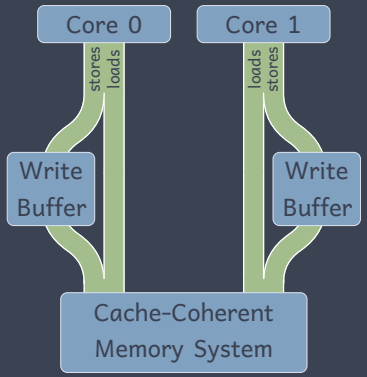
$$A <p\, B \Rightarrow A <m\, B$$







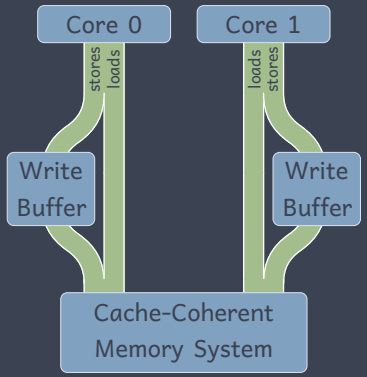
» Total Store Order



TSO/x86 memory model allows some non-SC executions by holding stores in write buffers:

* A store enters the write buffer immediately
* A store exits the write buffer when cache in a read-write coherence state

» Total Store Order



TSO/x86 memory model allows some non-SC executions by holding stores in write buffers:

* A store enters the write buffer immediately
* A store exits the write buffer when cache in a read-write coherence state

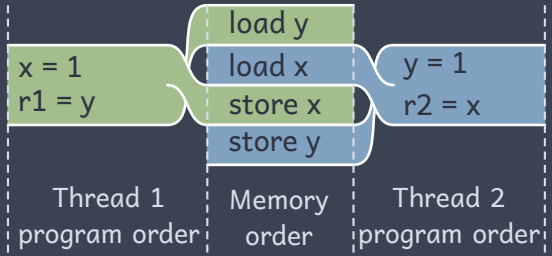
Thread 1

```
x = 1;
r1 = y;
```

Thread 2

```
y = 1;
r2 = x;
```

» Total Store Order



* $L(a) <p L(b) \Rightarrow L(a) <m L(b)$
* $L(a) <p S(b) \Rightarrow L(a) <m S(b)$
* $S(a) <p S(b) \Rightarrow S(a) <m S(b)$
* $S(a) <p L(b) \not\Rightarrow S(a) <m L(b)$
* Therefore it's possible to have both $r_1 = 0$ and $r_2 = 0$

» Program order

C++

```
x = y + 1;
y = 0;
```

```
g++ -O3 -S -masm=intel ctmo.cpp
```

ASM

```
mov eax, DWORD PTR Y[rip]
mov DWORD PTR Y[rip], 0
add eax, 1
mov DWORD PTR X[rip], eax
```

» Program order

C++

```
x = y + 1;
asm volatile ("" ::: "memory");
y = 0;
```

```
g++ -O3 -S -masm=intel ctmo.cpp
```

ASM

```
mov eax, DWORD PTR Y[rip]
add eax, 1
mov DWORD PTR X[rip], eax
mov DWORD PTR Y[rip], 0
```

» Program order

Thread 1

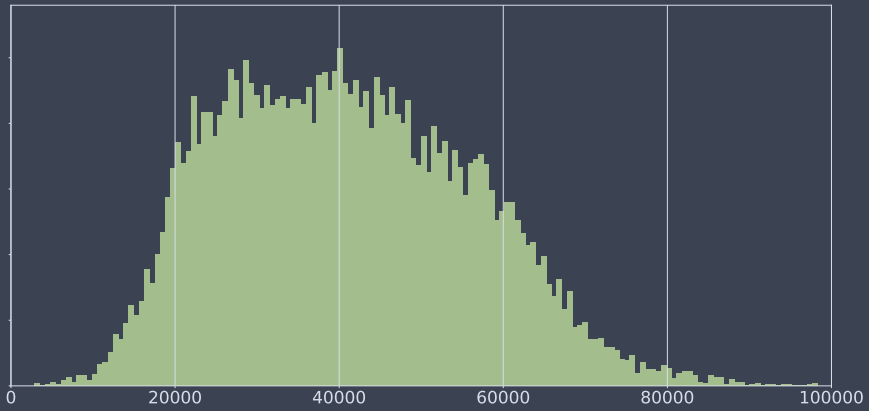
```
1 void thread_1_body ()
2 {
3   x = 1;
4   asm volatile (
5     "" ::: "memory");
6   r1 = y;
7 }
```

Thread 2

```
1 void thread_2_body ()
2 {
3   y = 1;
4   asm volatile (
5     "" ::: "memory");
6   r2 = x;
7 }
```

» Load after Store violations count

i7-7700k



» Program order

Thread 1

```
1 void thread_1_body ()
2 {
3   x = 1;
4   asm volatile (
5     "mfence" ::: "memory");
6   r1 = y;
7 }
```

Thread 2

```
1 void thread_2_body ()
2 {
3   y = 1;
4   asm volatile (
5     "mfence" ::: "memory");
6   r2 = x;
7 }
```

» Weak models

Weak memory models preserves only the orders that programmers require.

memory operations can be reordered unless there is a fence between them.

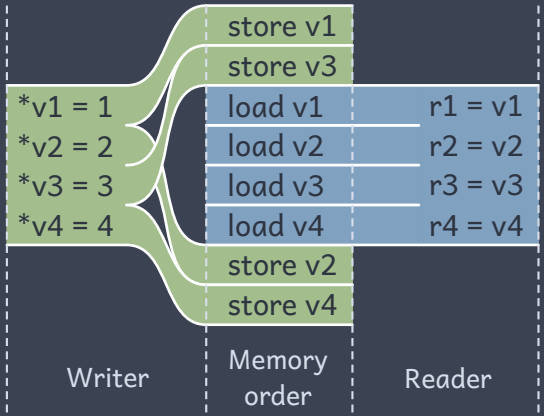
* $L(a) <p \text{ FENCE} \Rightarrow L(a) <m \text{ FENCE}$
* $S(a) <p \text{ FENCE} \Rightarrow S(a) <m \text{ FENCE}$
* $\text{FENCE} <p\, L(a) \Rightarrow \text{FENCE} <m\, L(a)$
* $\text{FENCE} <p\, S(a) \Rightarrow \text{FENCE} <m\, S(a)$

» Operations interleaving

```
__device__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2;
  *v3 = 3;
  *v4 = 4;
}
```

```
__device__ void reader (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4,
    int *result)
{
  int r1 = *v1; int r2 = *v2;
  int r3 = *v3; int r4 = *v4;

  if (   r1 == 1 && r3 == 3
      && r2 == 0 && r4 == 0)
    *result = 0;
}
```

» Operations interleaving



» Operations interleaving

Was interleaved up to **671** times out of 1000 runs on RTX 3090.

Writer

```
IMAD.MOV.U32 R0, RZ, RZ, 0x1;
IMAD.MOV.U32 R2, RZ, RZ, 0x2;
IMAD.MOV.U32 R3, RZ, RZ, 0x3;
IMAD.MOV.U32 R4, RZ, RZ, 0x4;
STG.E.STRONG.SYS [UR4], R0;
STG.E.STRONG.SYS [UR6], R2;
STG.E.STRONG.SYS [UR8], R3;
STG.E.STRONG.SYS [UR10], R4;
```

Reader

```
LDG.E.STRONG.SYS R0, [UR4];
LDG.E.STRONG.SYS R2, [UR6];
LDG.E.STRONG.SYS R3, [UR8];
LDG.E.STRONG.SYS R4, [UR10];
```

» Operations interleaving

```
cudaMemset (x, 0, 4 * sizeof (int));
cudaMemset (y, 0, 2 * sizeof (int));

if (single_segment)
  kernel<<<1024, 32>>> (x + 0, x + 1, x + 2, x + 3, r);
else
  kernel<<<1024, 32>>> (x + 0, y + 0, x + 1, y + 1, r);
```

» Write Buffer

```
__host__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{          // <=
  *v1 = 1;
  *v2 = 2;
  *v3 = 3;
  *v4 = 4;
}
```

Entry: single word

addr	value

» Write Buffer

```
__host__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1; // <=
  *v2 = 2;
  *v3 = 3;
  *v4 = 4;
}
```

Entry: single word

addr	value
200	1

» Write Buffer

```
__host__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2; // <=
  *v3 = 3;
  *v4 = 4;
}
```

Entry: single word

addr	value
200	1
400	2

» Write Buffer

```
__host__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2;
  *v3 = 3; // <=
  *v4 = 4;
}
```

Entry: single word

addr	value
200	1
400	2
204	3

» Write Buffer

```
__host__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2;
  *v3 = 3;
  *v4 = 4; // <=
}
```

Entry: single word

addr	value
200	1
400	2
204	3
404	4

» Merging Write Buffer

```
__device__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{          // <=
  *v1 = 1;
  *v2 = 2;
  *v3 = 3;
  *v4 = 4;
}
```

Entry: cache block

addr	value	value	value	value

» Merging Write Buffer

```
1  __device__ void writer (
2      volatile int * v1,
3      volatile int * v2,
4      volatile int * v3,
5      volatile int * v4)
6  {
7    *v1 = 1; // <=
8    *v2 = 2;
9    *v3 = 3;
10   *v4 = 4;
11 }
```

Entry: cache block

addr	value	value	value	value
200	1			

» Merging Write Buffer

```
__device__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2; // <=
  *v3 = 3;
  *v4 = 4;
}
```

Entry: cache block

addr	value	value	value	value
200	1			
400	2			

» Merging Write Buffer

```
__device__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2;
  *v3 = 3; // <=
  *v4 = 4;
}
```

Entry: cache block

addr	value	value	value	value
200	1	3		
400	2			

» Merging Write Buffer

```
1  __device__ void writer (
2      volatile int * v1,
3      volatile int * v2,
4      volatile int * v3,
5      volatile int * v4)
6  {
7    *v1 = 1;
8    *v2 = 2;
9    *v3 = 3;
10   *v4 = 4; // <=
11 }
```

Entry: cache block

addr	value	value	value	value
200	1	3		
400	2	4		

» Operations interleaving

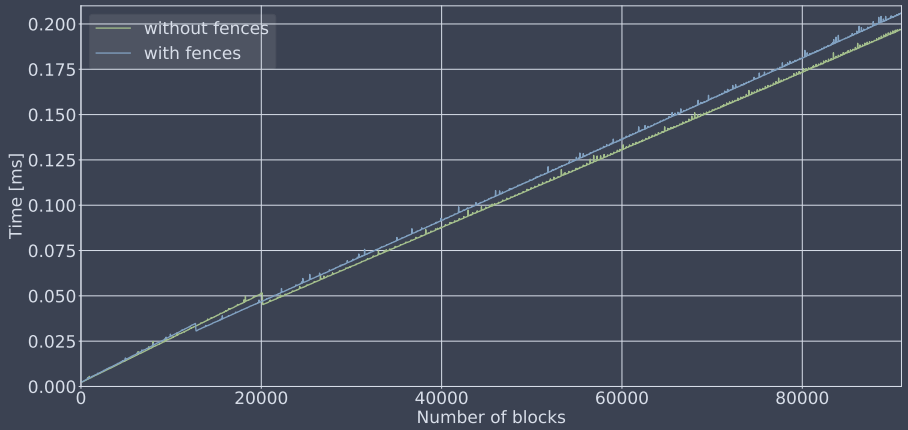
fixed version

```
__device__ void writer (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4)
{
  *v1 = 1;
  *v2 = 2;
  __threadfence ();
  *v3 = 3;
  *v4 = 4;
}
```

```
__device__ void reader (
    volatile int * v1,
    volatile int * v2,
    volatile int * v3,
    volatile int * v4,
    int *result)
{
  int r1 = *v1; int r2 = *v2;
  __threadfence ();
  int r3 = *v3; int r4 = *v4;

  if (   r1 == 1 && r3 == 3
      && r2 == 0 && r4 == 0)
    *result = 0;
}
```

» Operations interleaving

RTX 3090

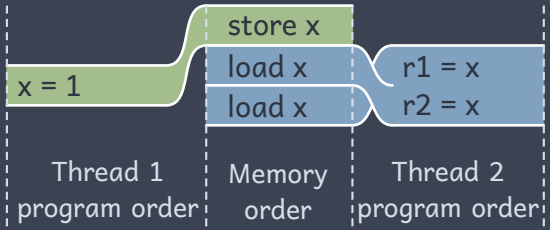


» Read-Read Coherence

coRR

```
__global__ void coherence_of_read_read (/*...*/) {
  __shared__ int cache[BLOCK_SIZE];
  /* .. Initialize shared memory .. */
  __syncthreads ();
  if (tid == wthread) {
    cache[tid] = 1;
  }
  else {
    const int r1 = cache[reordering_1[tid]];
    const int r2 = cache[reordering_2[tid]];
    if (r1 == 1 && r2 == 0)
      if (tid == rthread)
        *violated = 1;
  }
}
```

» Read-Read Coherence



» Read-Read Coherence

Violated **157** times out of 100’000 runs on GTX 560 (Fermi). Fixed in Maxwell.

Thread 1

```
MOV32I R0, 0x1;
STS [R7], R0;
```

Thread 2

```
LDS R0, [R0];
LDS R4, [R4]; /* [R4] = [R0] */
ISETP.EQ.AND P0, PT, R0, 0x1, PT;
ISETP.EQ.AND P0, PT, R4, RZ, P0;
```

» Message Passing

```
__global__ void kernel (
    int n, int * flag, int * data, int *result) {
```

Writer

```
for (int i = 0; i < n; i++)
  data[i] = i + 1;

*flag = 1;
```

Reader

```
while (*flag == 0);

for (int i = 0; i < n; i++)
  if (data[i] == 0)
    *result = 0;
```

» Message Passing

```
__global__ void kernel (
    int n, int * flag, int * data, int *result) {
```

Writer

```
for (int i = 0; i < n; i++)
  data[i] = i + 1;

*flag = 1;
```

SASS

```
IADD R3, R0, 0x1;
IADD R4, R0, 0x2;
STG.E [R1], R3
IADD R3, R0, 0x3;
STG.E [R2], R4;
; /* ... */
MOV32I R5, 0x1;
STG.E [flag], R5;
```

» Message Passing

```
__global__ void kernel (
    int n, int * flag, int * data, int *result) {
```

Reader

```
while (*flag == 0);

for (int i = 0; i < n; i++)
  if (data[i] == 0)
    *result = 0;
```

SASS

```
LDG.E R1, [R0];
ISETP.NE.AND P0, PT, R2, RZ, PT;
@!P0 BRA .L1;
LDG.E R3, [R2];
LDG.E R4, [R2+0x4];
LDG.E R5, [R2+0x8];
LDG.E R6, [R2+0xc];
; /* ... */
EXIT;
.L1:
BRA .L1;
```

» Message Passing

first fix

```
__global__ void kernel (
    int n, volatile int * flag, int * data, int *result) {
```

Reader

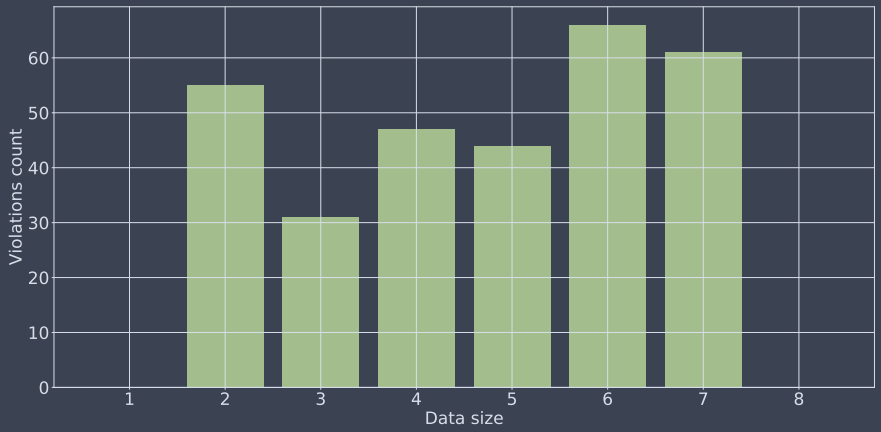
```
while (*flag == 0);

for (int i = 0; i < n; i++)
  if (data[i] == 0)
    *result = 0;
```

SASS

```
.L1:
LDG.E.STRONG.SYS R1, [R0];
ISETP.NE.AND P0, PT, R2, RZ, PT;
@!P0 BRA .L1;
LDG.E R3, [R2];
LDG.E R4, [R2+0x4];
LDG.E R5, [R2+0x8];
LDG.E R6, [R2+0xc];
; /* ... */
EXIT;
```

» Stale data found

RTX 2080



» Message Passing

final fix

```
__global__ void kernel (
    int n, volatile int * flag, int * data, int *result) {
```

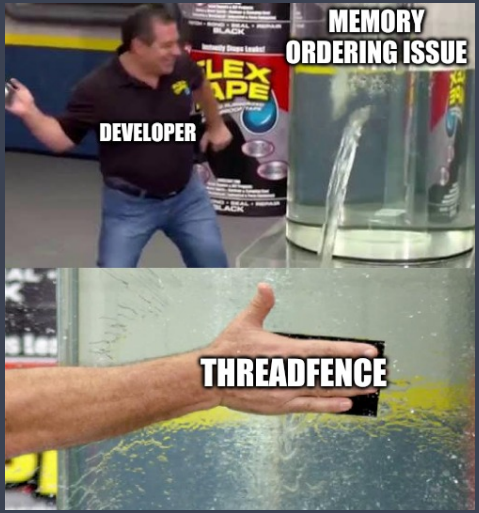
Writer

```
for (int i = 0; i < n; i++)
  data[i] = i + 1;
__threadfence ();

*flag = 1;
```

Reader

```
while (*flag == 0);
__threadfence ();

for (int i = 0; i < n; i++)
  if (data[i] == 0)
    *result = 0;
```

» Message Passing



» Message Passing

Controversial version

```
__device__ int ldg (const int * p)
{
  int out;
  asm volatile("ld.global.cg.s32 %0, [%1];" : "=r"(out) : "l"(p));
  return out;
}
```

Writer

```
for (int i = 0; i < n; i++)
  data[i] = i + 1;
__threadfence ();

*flag = 1;
```

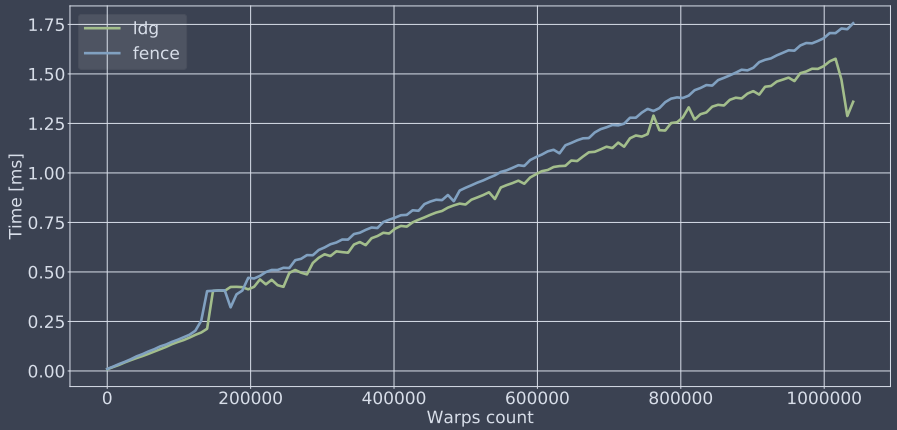
Reader

```
while (*flag == 0);
// __threadfence ();

for (int i = 0; i < n; i++)
  if (ldg (data + i) == 0)
    *result = 0;
```

» Message Passing

n=1

Average speedup is about 8.4%.



» Message Passing

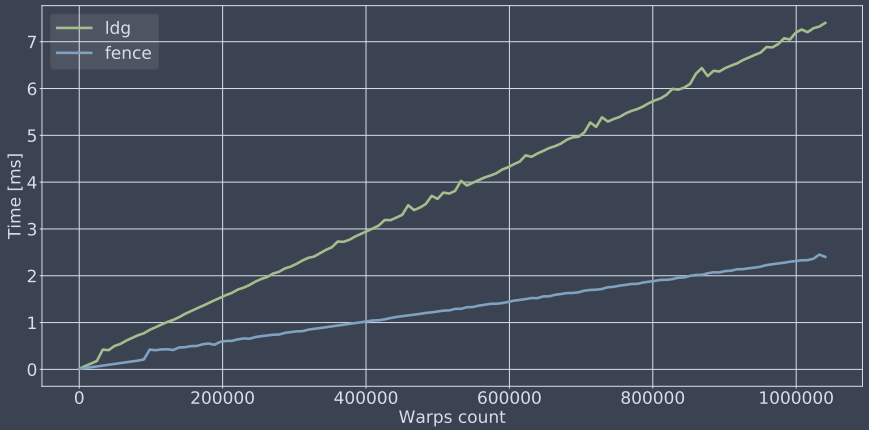
"It's possible to break the dependency order between the flag load and the later, dependent ld.cg load, that makes your test pass. Because it's possible to write a program that breaks this order, it's not possible for us to write a memory model that says the order is enforced."[3]

[3] Olivier Giroux. personal communication.

» Message Passing

n=9

Average slowdown is about 208.3%.



» Message Passing

The fence can be eliminated if the status flag and the corresponding value being updated can be combined into a single architectural word.[4]

[4] M. Garland D. Merrill. "Single-pass Parallel Prefix Scan with Decoupled Look-back". In: *NVIDIA Technical Report NVR-2016-002* (2016).

» Message Passing

Single word

```
union data_flag
{
  struct { int32_t data; int32_t flag; } fields;
  int64_t vec;
};
```

Writer

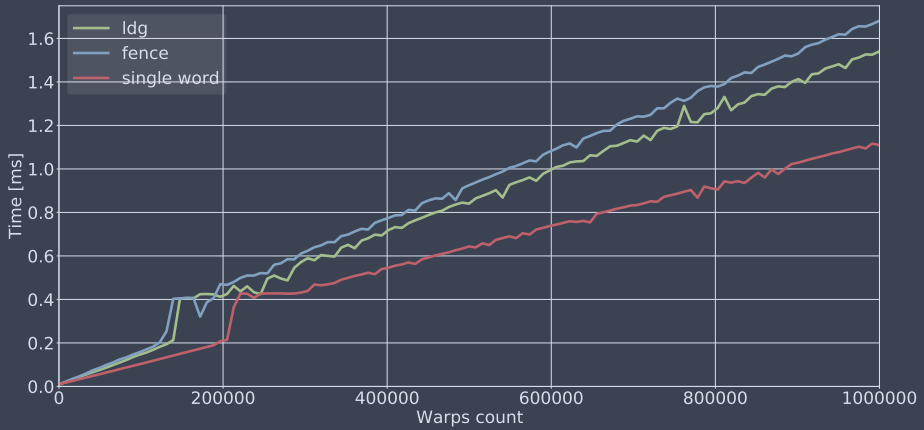
```
data_flag tmp;
tmp.fields.data = 1;
tmp.fields.flag = 1;

df->vec = tmp.vec;
```

Reader

```
while (!tmp.fields.flag)
  tmp.vec = df->vec;

if (tmp.fields.data == 0)
  *result = 0;
```

» Message Passing

Single word



» Exclusive scan

sequential version

Scan produces an output sequence where each element is computed to be the reduction of the elements occurring earlier in the input sequence.

```
int sum = 0;
for (int i = 0; i < n; i++)
  {
    out[i] = sum;
    sum += in[i];
  }
```

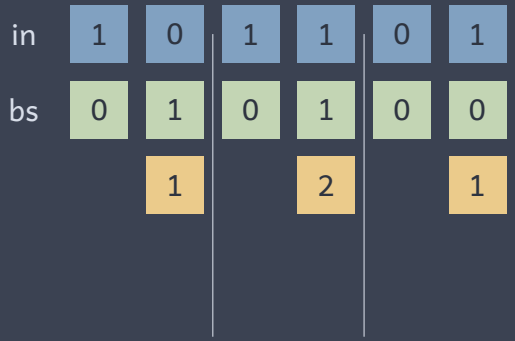
in	1	0	1	1	0	1
out	0	1	1	2	3	3

» Exclusive scan

in	1	0	1	1	0	1
bs	0	1	0	1	0	0

» Exclusive scan

blocks' results



» Exclusive scan

inclusive scan of blocks' results

in	1	0	1	1	0	1
bs	0	1	0	1	0	0
		1		2		1
		1		3		4

» Exclusive scan

final result

in	1	0	1	1	0	1
bs	0	1	0	1	0	0
		1		2		1
		1		3		4
out	0	1	1	2	3	3

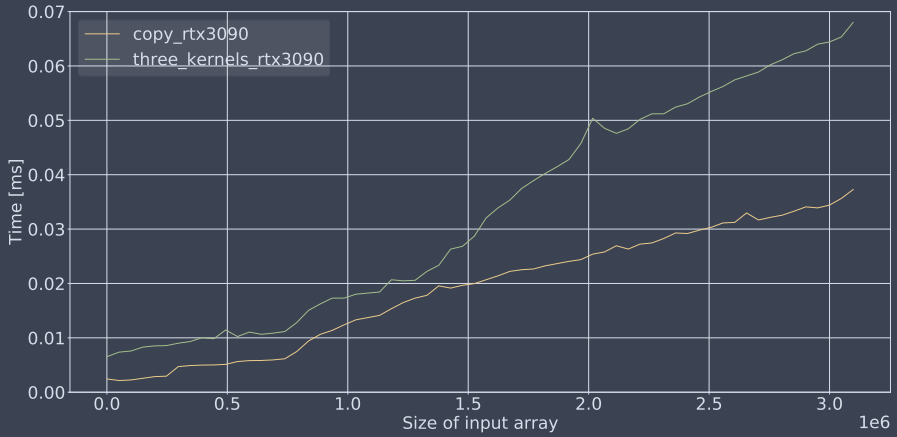
» Exclusive scan

three-kernels version

```
hierarchical_partial_scan<<<blocks, hierarchical_block_size>>> (
  in, out, helper_buffer);

hierarchical_blocks_results_scan<<<1, 1024>>> (
  blocks, helper_buffer, helper_buffer);

hierarchical_final_adjust<<<blocks, hierarchical_block_size>>> (
  out, helper_buffer);
```

» Exclusive scan

three-kernels scan performance



» Exclusive scan

sequential version

```
// Load
data_type thread_data[thread_data_size];

BlockLoad (temp_storage).Load (in + block_offset, thread_data);
__syncthreads ();

// This thread block aggregate
data_type block_reduce =
  BlockReduce (temp_storage).Sum (thread_data);

/// ...
```

» Exclusive scan

sequential version

```
if (threadIdx.x == 0) {
  if (blockIdx.x == 0) {
    commit (block_statuses + blockIdx.x, block_reduce);
  }
  else {
    do {
      prev_block_status = load (block_statuses + blockIdx.x - 1);
    } while (prev_block_status.flag == 0);

    thread_data[0] += prev_block_status.data;
    commit (
      block_statuses + blockIdx.x,
      prev_block_status.data + block_reduce);
  }
}
__syncthreads ();
```

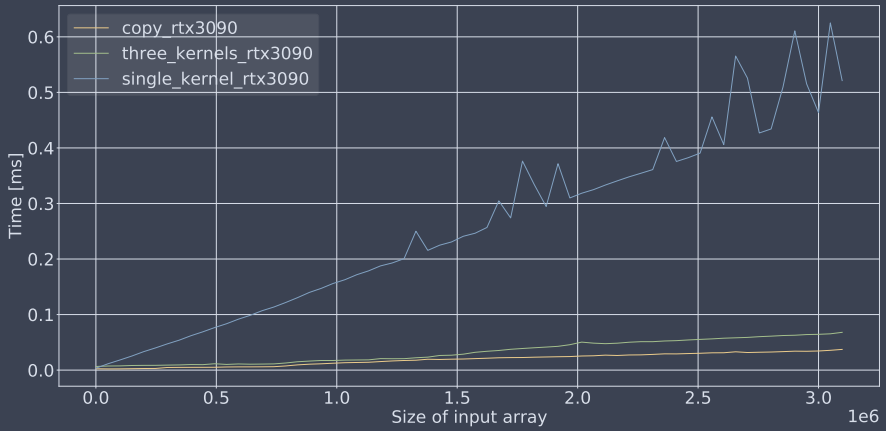
» Exclusive scan

sequential version

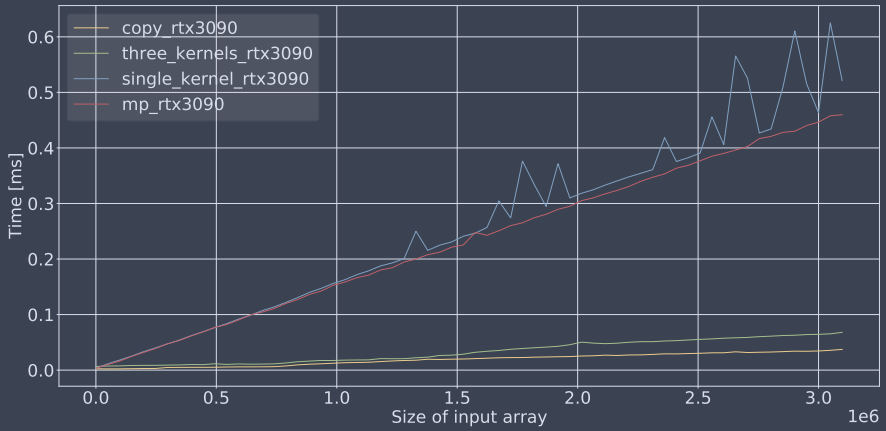
```
BlockScan (temp_storage).ExclusiveSum (thread_data, thread_data);
__syncthreads ();

// Store final result
BlockStore (temp_storage).Store (out + block_offset, thread_data);
```

» Exclusive scan

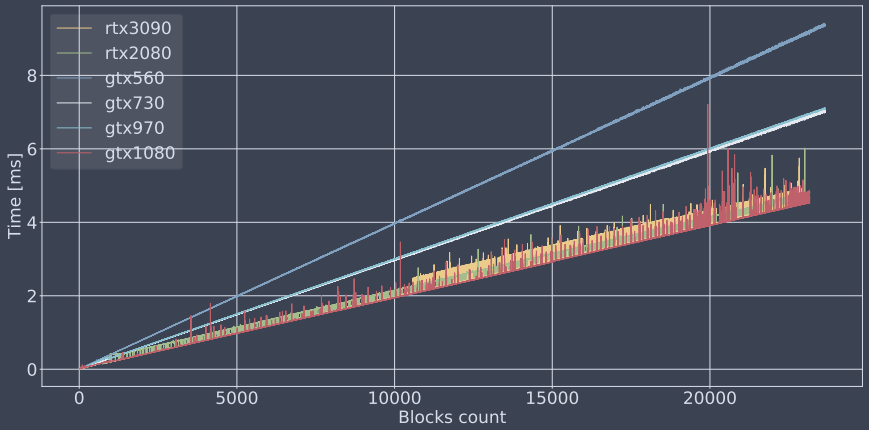
single scan performance



» Exclusive scan



» Message Passing



» Exclusive scan

```
if (blockIdx.x == 0)
  if (threadIdx.x == 0)
    store (block_statuses + blockIdx.x, 2, block_aggregate);
```

» Exclusive scan


```
if (blockIdx.x != 0)
{
  if (threadIdx.x == 0)
    store (block_statuses + blockIdx.x, 1, block_aggregate);

  if (threadIdx.x / WARP_SIZE == 0)
  {
    // ...
```

» Exclusive scan


```
int predecessor_idx = blockIdx.x - WARP_SIZE + threadIdx.x;
data_type thread_sum = data_type ();

while (true)
{
  prev_block_status = load (block_statuses + predecessor_idx);

  while (prev_block_status.flag == 0)
    prev_block_status = load (block_statuses + predecessor_idx);
```

» Exclusive scan

```
if (__all_sync (WARP_MASK, prev_block_status.flag < 2)) {
  thread_sum += prev_block_status.data;
  predecessor_idx -= WARP_SIZE;
  continue;
}
else {
  const int final_mask =
    __ballot_sync (WARP_MASK, prev_block_status.flag > 1);

  const int rightmost_thread = WARP_SIZE - 1 - __clz (final_mask);

  thread_sum += threadIdx.x % WARP_SIZE < rightmost_thread
              ? 0
              : prev_block_status.data;
  break;
}
```

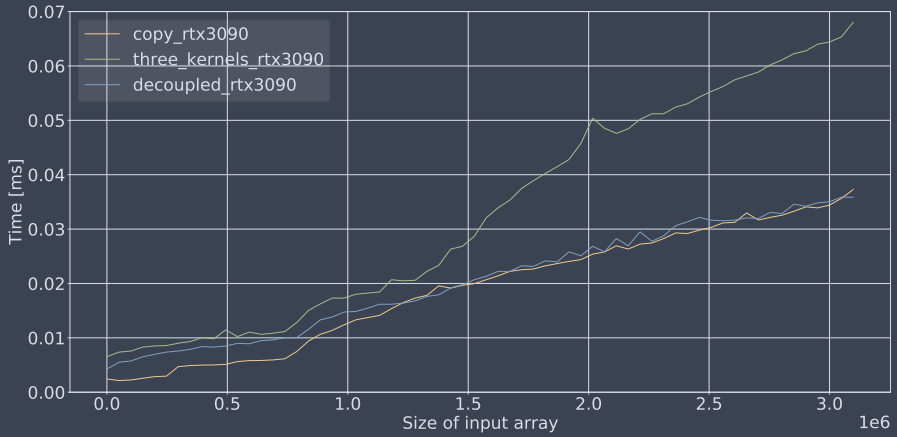
» Exclusive scan


```
data_type aggregate =
  WarpReduce (temp_storage.warp_reduce).Sum (thread_sum);

if (threadIdx.x == 0) {
  store (block_statuses + blockIdx.x, 2, aggregate + block_aggregate);
  prev_aggregate = aggregate;
}
```

» Exclusive scan

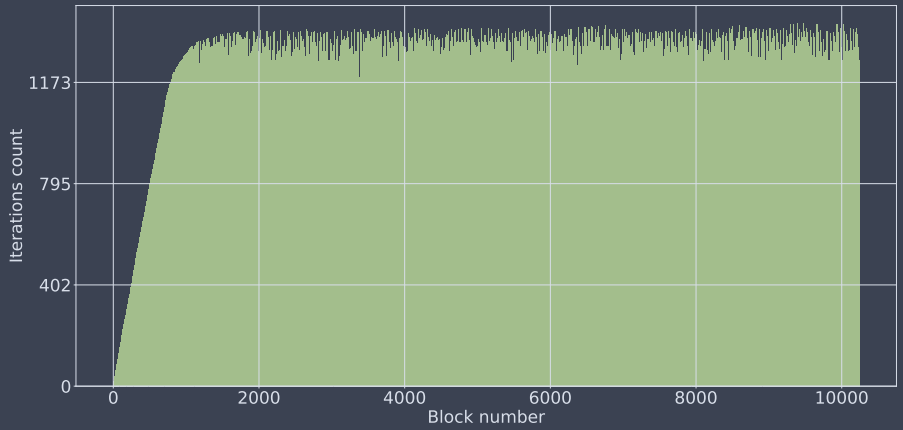
```
      // ...
    }
  }
__syncthreads ();

const data_type reg_prev_aggregate = prev_aggregate;
for (int i = 0; i < thread_data_size; i++)
  thread_data[i] += reg_prev_aggregate;

BlockStore (temp_storage).Store (out + block_offset, thread_data);
```

» Exclusive scan



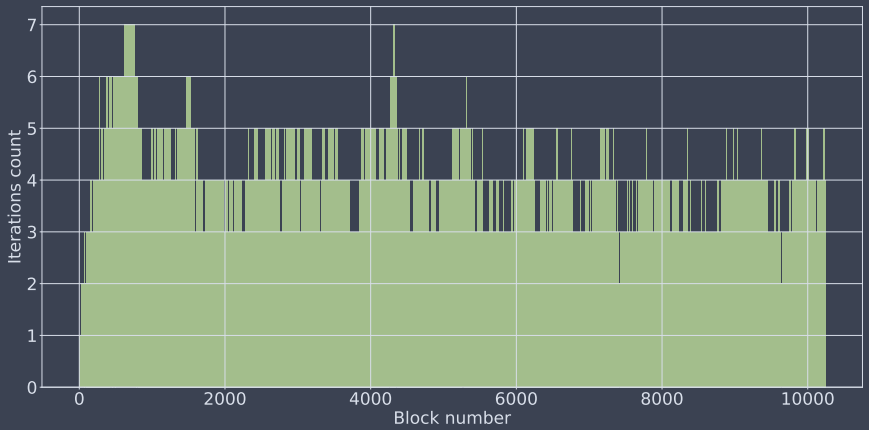
» Exclusive scan

single-kernel

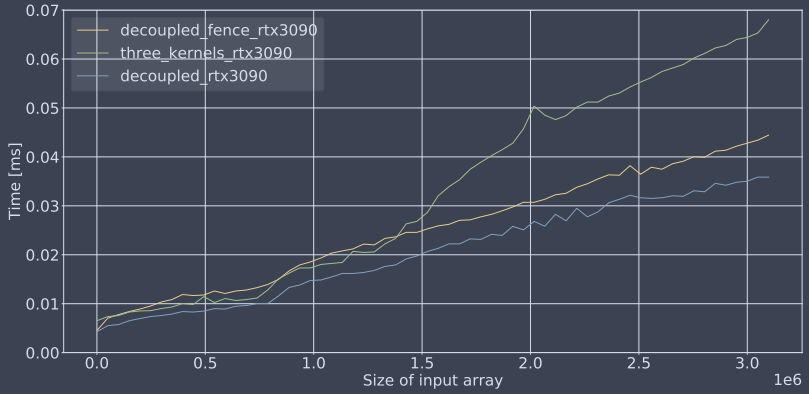


» Exclusive scan

decoupled look-back



Fence version: 23% performance degradation



» Release and Acquire Patterns

Release pattern on a location M consists of fence followed by a strong write on M in program order.

```
__threadfence ();
*flag = 1; // volatile int *flag
```

Acquire pattern on a location M consists of strong read on M followed by a fence in program order.

```
int reg = *flag; // volatile int *flag
__threadfence ();
```

» Definitions

Read operation All variants of ld instruction and atom instruction.

Write operation All variants of st instruction and atom instruction.

Memory operation A read or write operation.

» Definitions

Read operation	All variants of ld instruction and atom instruction.
Write operation	All variants of st instruction and atom instruction.
Memory operation	A read or write operation.
Strong operation	A fence operation, or a memory operation with a .relaxed, .acquire, .release, .acq_rel or .volatile qualifier.

» Release and Acquire Patterns

Release pattern on a location M consists of a release operation on M.

```
__device__ void store_release (int * flag, int val)
{
  asm volatile (
    "st.release.gpu.b32 [%0], %1;"
 :: "l"(flag),"r"(val) : "memory");
}
```

Acquire pattern on a location M consists of an acquire operation on M.

```
__device__ int load_acquire (int * flag)
{
  int reg;
  asm volatile (
    "ld.acquire.gpu.b32 %0, [%1];"
  : "=r"(reg) : "l"(flag) : "memory");
  return reg;
}
```

» Release and Acquire Patterns

st.release.gpu

```
MEMBAR.ALL.GPU
ST.E.STRONG.GPU [UR4], R0
```

ld.acquire.gpu

```
LD.E.STRONG.GPU R0, [UR4]
CCTL.IVALL
```

» Release and Acquire Patterns

libcu++

```
__device__ void old_writer (
    int n, volatile int * flag, int * data, int *result) {
  for (int i = 0; i < n; i++)
    data[i] = i + 1;

  __threadfence ();
  *flag = 1;
}
```

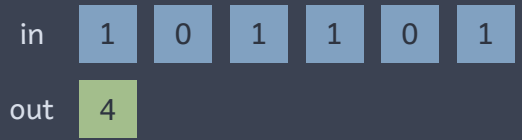
```
__device__ void new_writer (
    int n, atomic<int> &flag, int * data, int *result) {
  for (int i = 0; i < n; i++)
    data[i] = i + 1;

  flag.store (1, memory_order_release);
}
```

» Reduce

sequential version

Reduce produces *a* single *aggregate* from *a* list of input elements.

```
int sum = 0;
for (int i = 0; i < n; i++)
  sum += in[i];
*out = sum;
```



» Device reduction

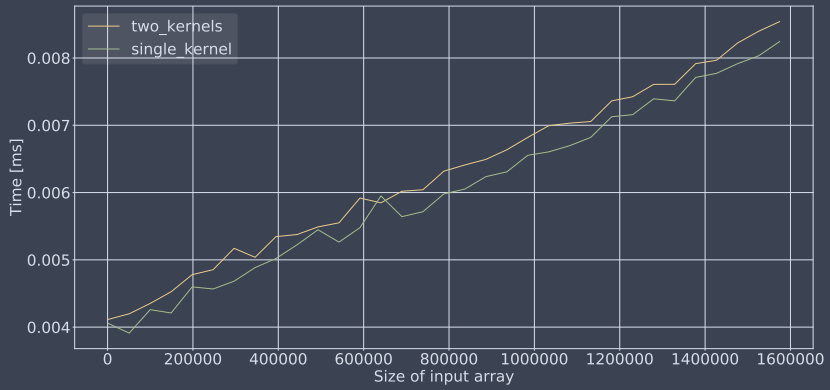
```
reduce<<<blocks, block_size>>> (in, block_results);
reduce_block_results<<<1, block_size>>> (
  blocks, block_results, block_results);
```

» Device reduction

```
// .. block-wide reduce ..

if (threadIdx.x == 0)
  {
    block_results[blockIdx.x] = block_result;
    __threadfence ();

    const int prev_count = atomicAdd (count, 1);
    temp_storage.need_to_perform_final_reduce
      = prev_count == gridDim.x - 1;
}
```

» Device reduction

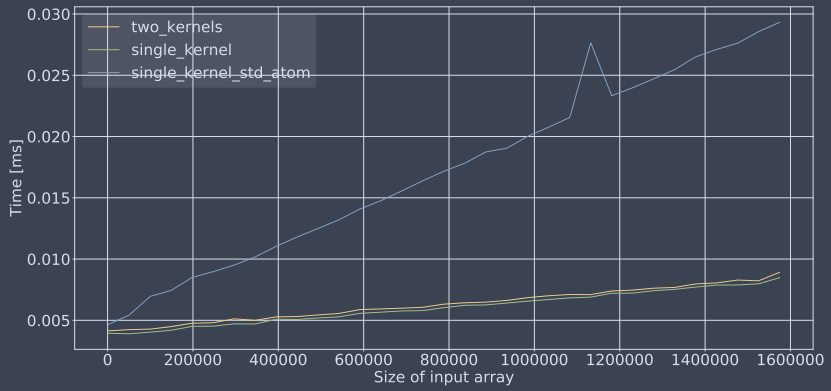
Single-kernel version: ≈ 4% performance improvement



» Device reduction

```
// .. block-wide reduce ..

if (threadIdx.x == 0)
  {
    block_results[blockIdx.x] = block_result;

    const int prev_count =
      count->fetch_add (1, cuda::std::memory_order_release);

    temp_storage.need_to_perform_final_reduce
      = prev_count == gridDim.x - 1;
}
```

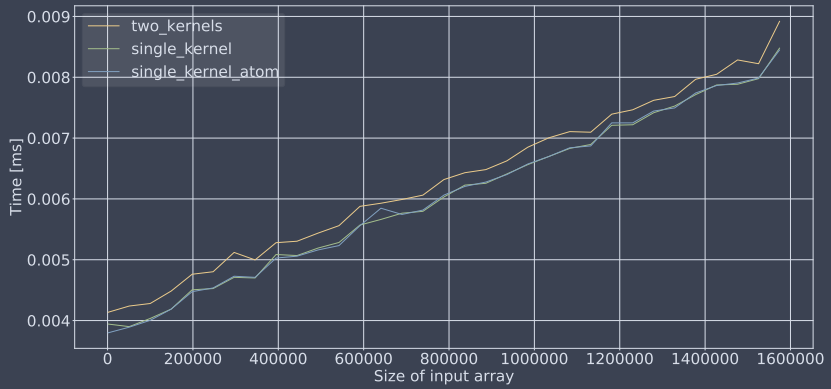
» Device reduction



» Device reduction

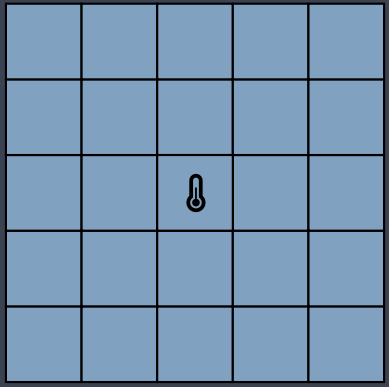
```cpp
// CUDA C++, __host__ __device__.
// Strictly conforming to the C++ Standard.
#include <cuda/std/atomic>
cuda::std::atomic<int> x;

// CUDA C++, __host__ __device__.
// Conforming extensions to the C++ Standard.
#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

» Device reduction



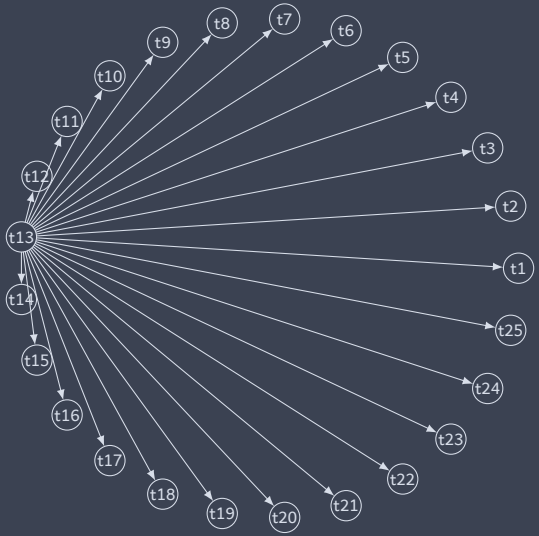
» Broadcast

illustration



» Broadcast

illustration



» Broadcast

basic version

```
unsigned int special_value = (unsigned int) -1;
unsigned int last_it = special_value;
int stride = blockDim.x * gridDim.x;


for (unsigned it = 0; it < last_it; it++) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  for (; i < n; i += stride) {
    data[i] += value;
    if (data[i] > threshold)
      stop = true;
  }

  if (stop)
    last_it = it;
}
```

» Broadcast

volatile global

sensor owner

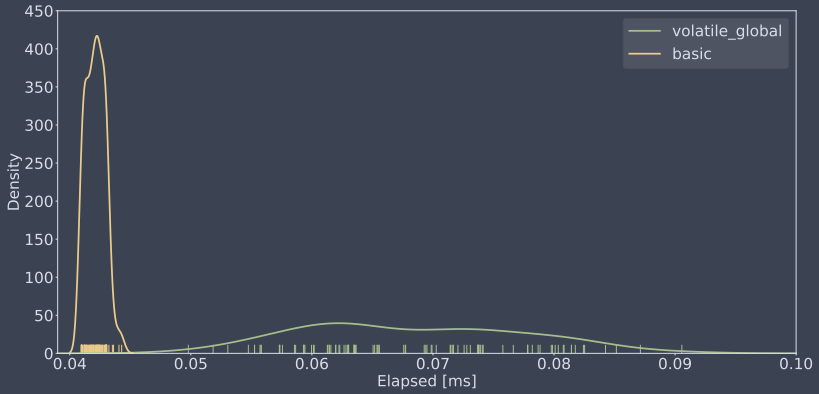
```
if (data[i] > threshold)
  stop = true;

store (flag, stop, it + 1);
```

other threads

```
if (special_value == last_it) {
  status_word it_state
    = load (flag);

  while (it_state.data <= it)
    it_state = load (flag);

  if (it_state.flag)
    last_it = it_state.data;
}
```

» Broadcast

volatile global performance



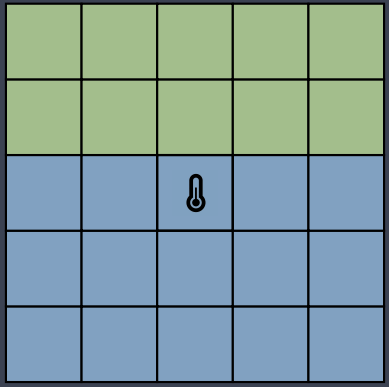
» Broadcast

volatile global block

```
if (last_it == special_value) {
  if (threadIdx.x == 0) {
    status_word it_state = load (flag);

    while (it_state.data <= it)
      it_state = load (flag);

    if (it_state.flag)
      block_last_it = it_state.data;
  }
  __syncthreads ();

  last_it = block_last_it;
}
```

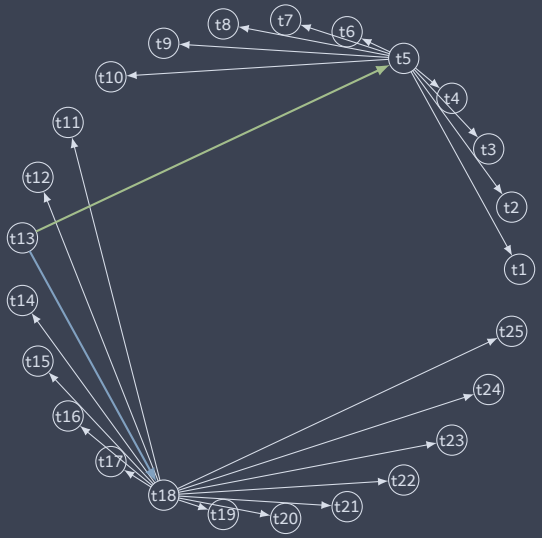
» Broadcast

illustration



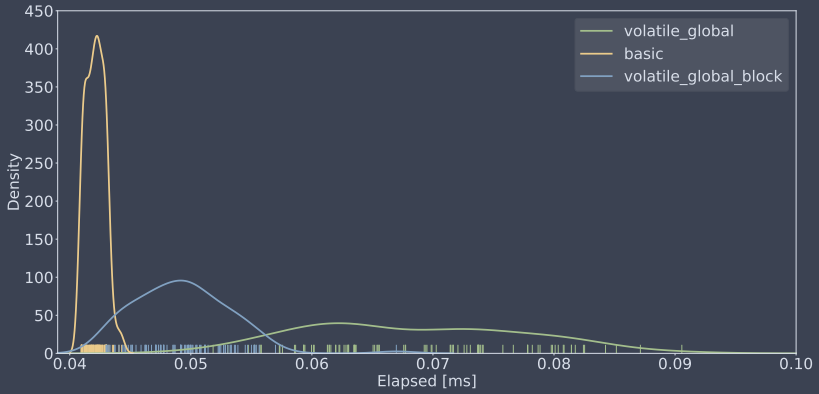
» Broadcast

illustration



» Broadcast

volatile global block performance



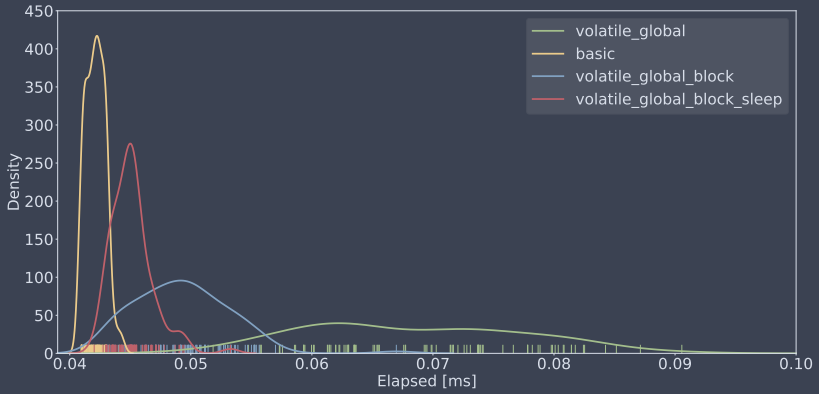
» Broadcast

volatile global block sleep

```
if (last_it == special_value) {
  if (threadIdx.x == 0) {
    status_word it_state = load (flag);

    while (it_state.data <= it) {
      __nanosleep (250); //< new line
      it_state = load (flag);
    }

    if (it_state.flag)
      block_last_it = it_state.data;
  }
  __syncthreads ();

  last_it = block_last_it;
}
```

» Broadcast

volatile global block sleep performance



» Broadcast

atomic (sensor owner)

```
if (data[i] > threshold)
  stop = true;

flag->store (
  status_word {stop, it + 1},
  cuda::memory_order_relaxed);
flag->notify_all ();
```

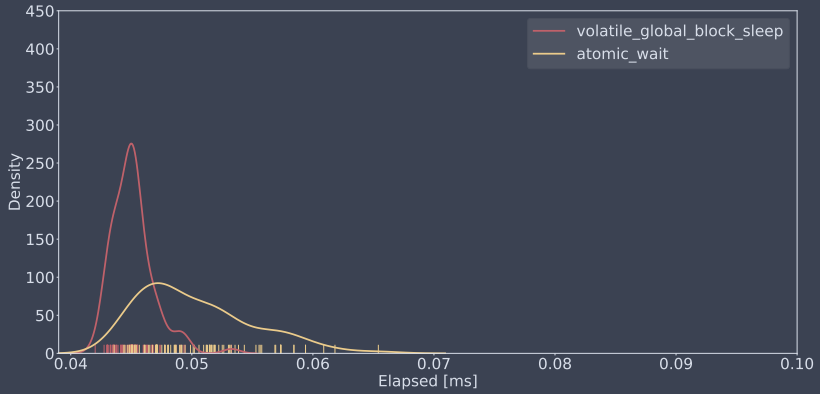
» Broadcast

atomic

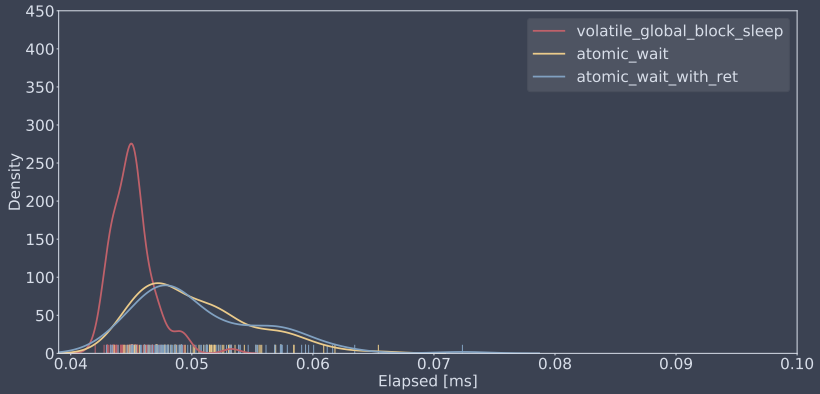
other threads

```
if (threadIdx.x == 0) {
  status_word it_state
    = flag->load (cuda::memory_order_relaxed);

  if (it_state.data <= it) {
    flag->wait (it_state, cuda::memory_order_relaxed);
    it_state = flag->load (cuda::memory_order_relaxed);
  }

  if (it_state.flag)
    block_last_it = it_state.data;
}
__syncthreads ();

last_it = block_last_it;
```

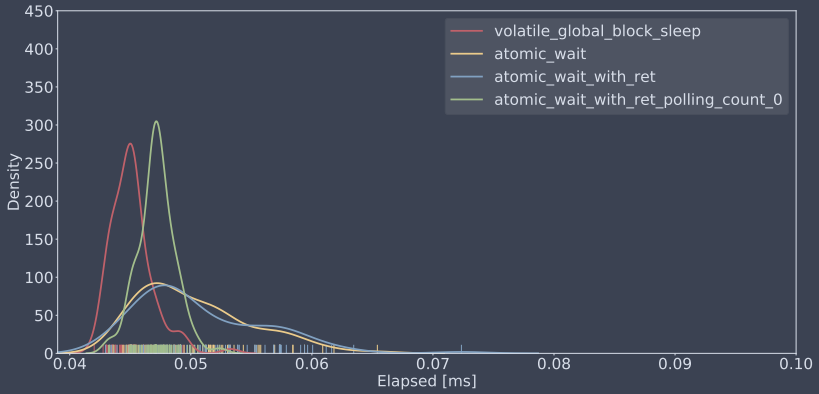
» Broadcast


volatile_global_block_sleep
atomic_wait
Density
Elapsed [ms]
0
50
100
150
200
250
300
350
400
450
0.04
0.05
0.06
0.07
0.08
0.09
0.10

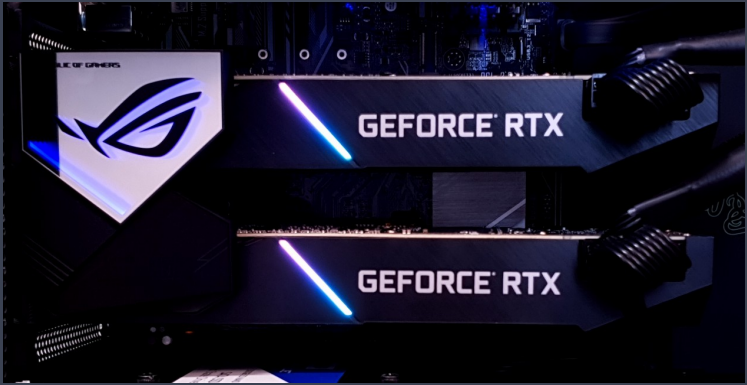
» Broadcast


volatile_global_block_sleep
atomic_wait
atomic_wait_with_ret
Density
450
400
350
300
250
200
150
100
50
0
0.04
0.05
0.06
0.07
0.08
0.09
0.10
Elapsed [ms]

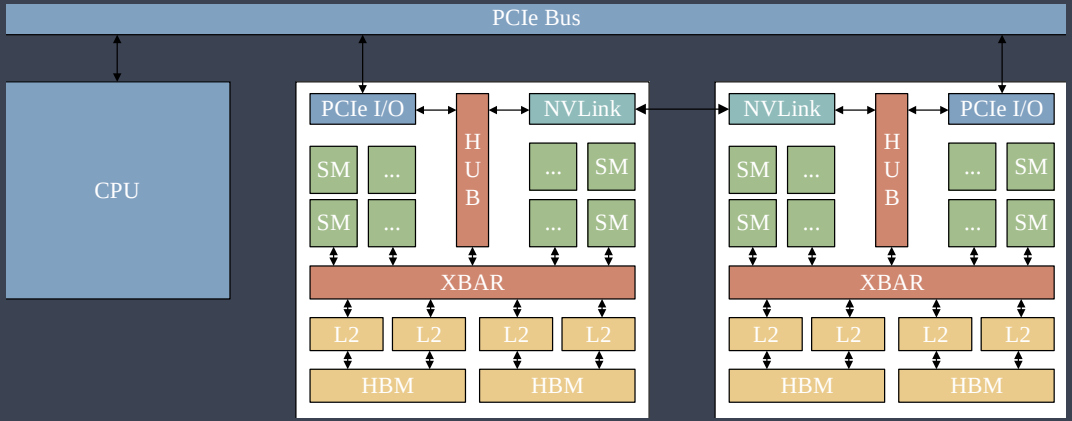
» Broadcast


volatile_global_block_sleep
atomic_wait
atomic_wait_with_ret
atomic_wait_with_ret_polling_count_0
Density
Elapsed [ms]
0
50
100
150
200
250
300
350
400
450
0.04
0.05
0.06
0.07
0.08
0.09
0.10

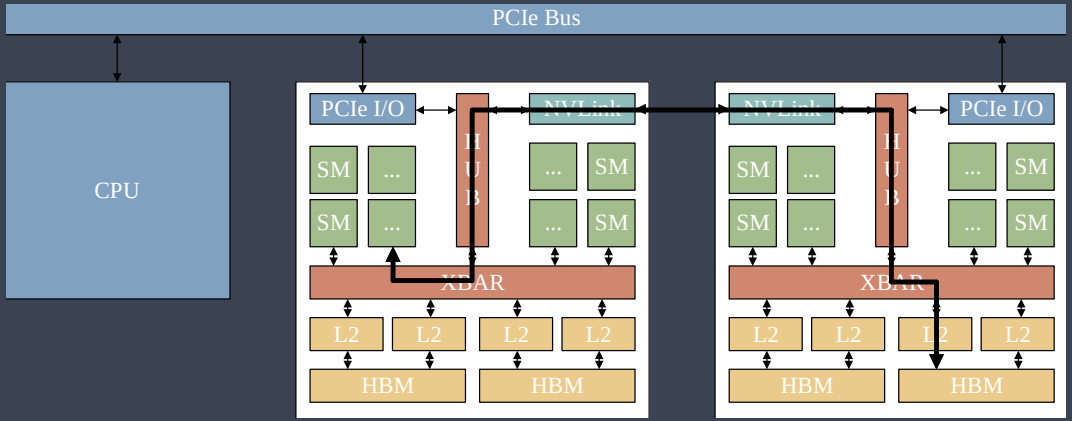
» Broadcast



» Broadcast

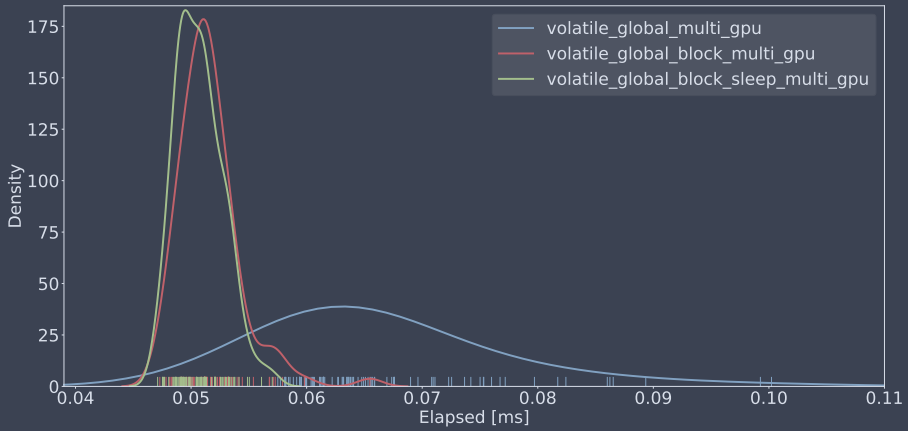

PCIe Bus
CPU
PCIe I/O
NVLink
HUB
SM
...
XBAR
L2
HBM
NVLink
PCIe I/O
HUB
SM
...
XBAR
L2
HBM

» Broadcast


PCIe Bus
CPU
PCIe I/O
NVLink
HUB
SM
...
XBAR
L2
HBM
NVLink
HUB
PCIe I/O
SM
...
XBAR
L2
HBM

» Broadcast

multi-GPU



Recommended materials

- CppCon 2018: Olivier Giroux “High-Radix Concurrent C++.”
- CppCon 2019: Olivier Giroux “The One-Decade Task: Putting std::atomic in CUDA.”
- Cpp Toronto 2020: Bryce Adelstein Lelbach “The CUDA C++ Standard Library by.”
- Sorin, Daniel J. and Hill, Mark D. and Wood, David A. “A Primer on Memory Consistency and Cache Coherence”

Source codes and presentation will be published soon, check @g_evtushenko