SYNCHRONIZATION IS BAD, BUT IF YOU MUST... (S9329)

Olivier Giroux, Distinguished Architect, ISO C++ Chair of Concurrency & Parallelism.
My coordinates

ogiroux@nvidia.com

Architects

NVIDIA GPU Engineers

Memory Model Community

WG21

ISO C++ Users
WHAT THIS TALK IS ABOUT:

- cudaDeviceSynchronize()
- __syncthreads()
- __shfl_l_sync()

✅ Using atomics to do blocking synchronization.
PSA: DON’T RUN SERIAL CODE IN THREADS

All blocked and no play makes t6 a dull thread…
PSA: RARE CONTENTION IS FINE
UNCONTENDED EXCHANGE LOCK

struct mutex { // suspend atomic<> disbelief for now

__host__ __device__ void lock() {
    while(1 == l.exchange(1, memory_order_acquire))
    ;
}

__host__ __device__ void unlock() {
    l.store(0, memory_order_release);
}

atomic<int> l = ATOMIC_VAR_INIT(0);
};

Awesome.

🎉 Thanks for attending my talk. 🎉
SIMT ATOMIC CONCERN SCALE:

😊 Atomic result feeds branch, closes loop, Volta+
🚄 Atomic result feeds branch, closes loop
😬 Atomic result feeds branch, inside loop
🎵 Atomic result feeds branch, outside loop
🧐 Atomic result feeds arithmetic
😁 Atomic result ignored
👶 No atomics
SIMT FAMILY HISTORY

- **Pixar CHAP**
  - Scalar channel programs.
- **Tesla SIMT**
  - Scalar thread programs.
  - Forward-progress = Nope 😞
- **Volta SIMT**
  - Scalar thread programs.
  - Forward-progress = YES! 🤘

**Time zero.**

APPLICABILITY
## SYNCHRONIZATION DECISION CHECKLIST

**CONs:**

1. Serialization is bad.
2. Critical path / Amdahl’s law.
3. Latency is high.

**PROs**

1. Algorithmic gains.
2. Latency hiding.
3. Throughput is high

**TL;DR:** Sometimes, it’s a win.
Keep local state in registers & shared memory, with synchronization.

See Greg Diamos’ GTC 2016 talk for more.
APP #2: LOCK-FREE IS NOT ALWAYS FASTER

// *continue* to suspend atomic<> disbelief for now
__host__ __device__ bool lock_free_writer_version(atomic<int>& a, atomic<int>& b) {
    int expected = -1;
    if(a.compare_exchange_strong(expected, 1, memory_order_relaxed))
        b.store(1, memory_order_relaxed);
    return expected == -1;
}

// This version is a ~60% speedup at GPU application level, despite progress hazards.
__host__ __device__ bool starvation_free_writer_version(atomic<int>& a, atomic<int>& b) {
    int expected_a = -1,
        expected_b = -1;
    bool success_a = a.compare_exchange_strong(expected_a, 1, memory_order_relaxed),
        success_b = b.compare_exchange_strong(expected_b, 1, memory_order_relaxed);
    if(success_a) // Note: we almost always succeed at both.
        while(!success_b) // <-- This loop makes this a deadlock-free algorithm.
            success_b = b.compare_exchange_strong(expected_b = -1, 1, memory_order_relaxed);
    else if(success_b)
        b.store(-1, memory_order_relaxed);
    return expected_a == -1;
}
Even if **mutexes** hide in every node, GPUs can build tree structures fast.

For more, see my CppCon 2018 talk on YouTube, *and* ‘Parallel Forall’ blog post.
PRE-REQUISITES
PR #1: FORWARD-PROGRESS

Every thread succeeds.

Some thread succeeds.

Wait-free

Obstruction-free

Starvation-free

Lock-free

Clash-free

App #2.

No limitations on thread delays

threads delayed infinitely often

thread delays limited

uniformly isolating scheduler

fair scheduler

No scheduling requirements.

(Any thread scheduler.)

Eventually run isolated.

Critical sections eventually complete.

PR #2: MEMORY CONSISTENCY

Classic CUDA C++

## PR #3: TRUE SHARING

- Concurrent data sharing between CPU and GPU is a new possibility.
- Real usefulness has some more conditions.

<table>
<thead>
<tr>
<th>Platform / allocator</th>
<th>Load/store sharing</th>
<th>Atomic (high cont’n)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Any</strong>: ARM/Windows/Mac/Unmanaged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x86 Linux (CPU/GPU) Managed</td>
<td>Yes.</td>
<td>Technically… but no.</td>
</tr>
<tr>
<td>x86 Linux (GPU/GPU) Managed</td>
<td>YES! TRY IT!</td>
<td></td>
</tr>
<tr>
<td>POWER Linux (all pairs) Managed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
__host__ __device__ void test(int my_thread, int total_threads, int final_value) {
    for(int old; my_thread < final_value; start += total_threads)
        while(!a.compare_exchange_weak(old = my_thread, my_thread + 1, memory_order_relaxed))
            ;
}
CONTENDING PROCESSORS ARE CRUSHED...

½ millisecond
...UNLESS THE PROCESSORS ARE NVLINK’ED.
ALL OF THE FOLLOWING SLIDES ARE NVLINK'ED.

And not log scale, because it’s legible in linear scale now. Thanks.
CONTENDED MUTEXES
CONTENDED MUTEXES
AS AN EXERCISE TO THINK ABOUT THROUGHPUT AND FAIRNESS
CONTENDED EXCHANGE LOCK

```c
struct mutex {

  __host__ __device__ void lock() {
    while (1 == l.exchange(1, memory_order_acquire))
    ;
  }

  __host__ __device__ void unlock() {
    l.store(0, memory_order_release);
  }

atomic<int> l = ATOMIC_VAR_INIT(0);
};
```

Not awesome.

ANTLR  Stay. Keep attending my talk. ANTLR
CONTENDED EXCHANGE LOCK

Heavy system pressure:
• A lot of requests
• Each request is slow
• **K** bounds forecast relative error (orange line):
  \[ \text{Latency}_{\text{response}} > K_{\text{delay}} \times \text{Latency}_{\text{impulse}} \]
  • Pick arbitrary \( K_{\text{delay}} \): say 1.5 for 50% error.
  • Some benefit to stochastic choice, avoid coupling.

• **Ceiling** trades bandwidth & maximum error:
  \[ \frac{t_{\text{polling}}}{(\text{lat}_{\text{loaded}} + \text{lat}_{\text{backoff}})} > \text{BW}_{\text{polling}} \]
  • Pick arbitrary \( \text{BW}_{\text{polling}} \): say 0.5 * \( \text{Bw}_{\text{contended}} \)

• **Floor** protects the fast corner (green box):
  \[ \text{Latency}_{\text{response}} > \text{Latency}_{\text{floor}} \]
  • Minimum CPU sleep (Linux) is \( \approx 50000\text{ns} \).
  • Minimum sleep on V100 is \( \approx 0\text{ns} \).
We seem to have fixed the slow corner.
FAST LOCKS, SLOW APPLICATIONS 🤔

- Fast because: lock *disproportionally* granted to some threads.
- Slow because: top-level performance *often* depends on fairness.

Single-thread rate is a strong attractor.
On the nature of progress.

Critical sections eventually complete.

RECALL: FORWARD-PROGRESS

Some thread succeeds.

- Wait-free
- Obstruction-free
- Starvation-free
- Lock-free
- Clash-free
- Deadlock-free

No limitations on thread delays
threads delayed infinitely often
thread delays limited
uniformly isolating scheduler
fair scheduler

CONTENDED EXCHANGE
LOCK WITH BACKOFF
WHEN IS DEADLOCK-FREE SUITABLE?

Enumerated list:
1. Very low contention.
2. Top-level algorithms resilient to tail effects.

Luckily, this is still pretty common!
```c
struct alignas(128) ticket_mutex {
    __host__ __device__ void lock() {
        auto const my = in.fetch_add(1, memory_order_acquire);
        while(1) {
            auto const now = out.load(memory_order_acquire);
            if(now == my)
                break;
            auto const delta = my - now;
            auto const delay = (delta << 8); // * 256
            #ifdef __CUDA_ARCH__
                __nanosleep(delay);
            #else
                if(delay > (1<<15)) // 32us
                    std::this_thread::sleep_for(std::chrono::nanoseconds(delay));
                else
                    std::this_thread::yield();
            #endif
        }
    }

    __host__ __device__ void unlock() {
        out.fetch_add(1, memory_order_release);
    }
}

atomic<unsigned> in = ATOMIC_VAR_INIT(0);
atomic<unsigned> out = ATOMIC_VAR_INIT(0);
```

Don’t need either K or ceiling here, delta is an accurate forecast! 😊
TICKET LOCK + PROPORTIONAL BACKOFF

![Graph showing latency in seconds for different numbers of threads and sections. The x-axis represents threads (GPU x CPU), the y-axis represents sections, and the z-axis represents latency (seconds). The graph shows a significant increase in latency as the number of threads increases.](image-url)
AGAIN : FORWARD-PROGRESS

Every thread succeeds.

Critical sections eventually complete.

WHEN IS STARVATION-FREE SUITABLE?

This is your default, when deadlock-free is unsuitable.

WHAT ELSE IS THERE FOR MUTEXES?

Wish we could use queue locks (e.g. MCS) but we can’t. These use O(P) storage 🤣 and local stack pointers (MCS).
BARRIERS
BARRIERS
AS A TYPICALLY-GPU THING
AND ALSO TO THINK ABOUT LATENCY
__host__ __device__ void arrive_and_wait() {

    auto const _expected = expected;
    auto const old = phase_arrived.fetch_add(1, memory_order_acq_rel);
    auto current = old + 1;

    if((old & phase_bit) != (current & phase_bit)) {
        phase_arrived.fetch_add(phase_bit - _expected);
    } else {
        while(1) {
            current = phase_arrived.load(memory_order_acquire);
            if((old & phase_bit) != (current & phase_bit))
                break;
            auto const delta = phase_bit - (current & ~phase_bit);
            auto const delay = (delta << 8); // * 256
            #ifdef __CUDA_ARCH__
                __nanosleep(delay);
            #else
                if(delay > (1ll<<15)) // 32us
                    std::this_thread::sleep_for(std::chrono::nanoseconds(delay));
                else
                    std::this_thread::yield();
            #endif
        }
    }
}

uint32_t const expected = 0;
atomic<uint32_t> phase_arrived = ATOMIC_VAR_INIT(0);
Centralized barrier is bad for the CPU.
- Coherence protocols strongly prefer fancy barrier algorithms: tree, tournament, dissemination...
- Because: $BW_{contended} = 1/Lat_{NUMA}$.

GPU just hangs-on for a while longer.
- But: fancy algorithms introduce high-latency, levels of indirection.
- Each indirection needs 1:100x .. 1:1000x improvement in BW to justify itself.
EASY AND EFFECTIVE GPU TREE BARRIER

- 2nd level of hierarchy is ~free, in blocks.
- Up to 1:1024 bandwidth reduction!

```c
__host__ __device__ void arrive_and_wait() {
#ifdef __CUDA_ARCH__
    auto const c = __syncthreads_count(1);
    if(threadIdx.x == 0)
        __arrive_and_wait(c);
    __syncthreads();
#else
    __arrive_and_wait();
#endif // __CUDA_ARCH__
}

__host__ __device__ void __arrive_and_wait(int c = 1) {
    auto const _expected = expected;
    auto const old = phase_arrived.fetch_add(c, memory_order_acq_rel);
    auto current = old + c;
    //...
}
```
“Remember, if you actually need a GPU barrier, then you should use cooperative groups instead.”

https://devblogs.nvidia.com/cooperative-groups/

- My inner CUDA engineer voice.
WHAT ABOUT CPU-GPU BARRIERS, THOUGH?

• As you can see, a new barrier algorithm is necessary.

• Perhaps partitioned strategies, by processor type?

Seriously, I’m asking. Somebody should try it! 😊

• I don’t know what it would be for, though. So no rush.
WHAT ELSE IS THERE FOR BARRIERS?

For multi-GPU systems:
• You can replicate arrivals to trade atomics vs. polling.
• Not done by CG but it has been done at NVIDIA.

For a DGX-2 (2.6 million threads):
• You *might* benefit from 3rd level of barrier, barely.
• I don’t think it’s been done at NVIDIA yet.
IN SHORT
1. Contention bandwidth is a major issue for synchronization. See: atomic story.

2. If you use back-offs, keep an eye on fairness. See: mutex story.

3. If you use indirection, the GPU needs a 100..1000x saving. See: barrier story.
CUDA::STD::ATOMIC<T> IS COMING SOON

Should come to the CUDA C++ toolkit this year, in 2019.

A preview is here: 
https://github.com/ogiroux/freestanding.

My CppCon 2018 talk has more, stream it on YouTube.
Concurrency at this scale has never been easier.

If you have IBM + V100 systems, try new algorithms!

We want to see what you’ll do with them.