Atomic Reductions

Gonzalo Brito Gadeschi

2024-10-23 @ 2024 LLVM Developers' Meeting

atomicrmw instruction

Atomic Read-Modify-Write

```
atomic<int> x = 0;
old = x.fetch_add(1, memory_order_relaxed);
```

```
%old = atomicrmw add ptr %p, i32 %value monotonic
```

```
atomically {
  %old = load %p
  %next = add %old, %value
  store ptr %p, %next
  ret %old;
}
```

Parallel Histogram Construction

Does not access value returned by atomicrmw

```
array<atomic<int>, N> buckets; int width;
T* data; int sz;


for_each(execution::par, views::iota(0, nthreads), [&](int t) {
  int nelem = sz / nthreads;
  for (int i = nelem * t; i < nelem * (t + 1); ++i)
      buckets[data[i] / width].fetch_add(1, memory_order_relaxed);
});
```

Hardware Support for Atomic Reductions

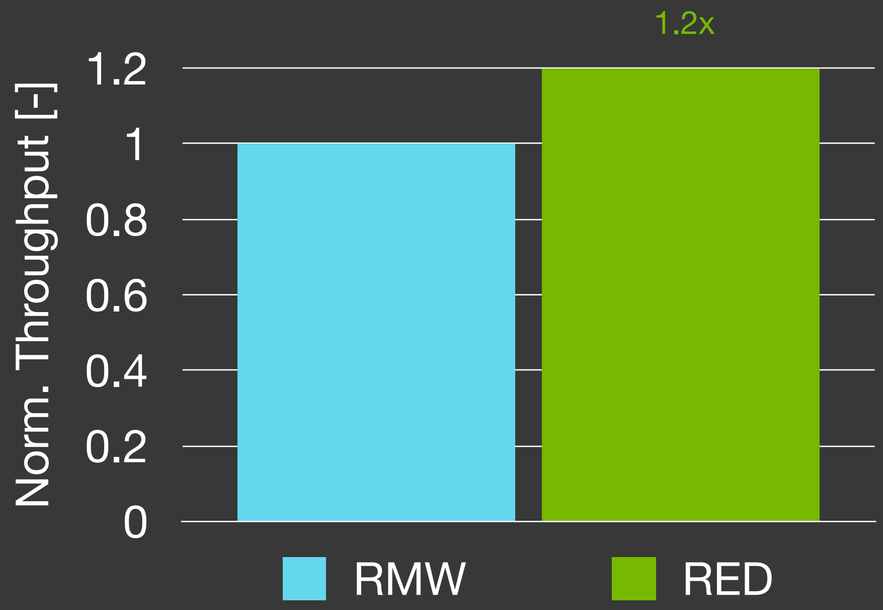
Atomic Read-Modify-Write Operations that “don’t return old”

- Arm: STADD/STUMIN/…, LDADD/SWP/CAS with “zero register” destination.
- PTX: red.<op> .
- x86 Remote Atomic Operations (RAO): AADD, AAND, AOR, AXOR.

Why? Performance

Histogram throughput with PTX “atom” vs PTX “red” on H100

```
for_each(execution::par, views::iota(0, nthreads),
 [&](int t) {
  int nelem = sz / nthreads;
  for (int i = nelem * t; i < nelem * (t + 1); ++i)
      buckets[data[i] / width]
          .fetch_add(1, memory_order_relaxed);
 }
);
```



Can LLVM optimize RMW into RED?

In some cases, it's unsound… hard to tell these apart.

```
atomic_int x = 0, y = 0, z = 0;

void thread0(atomic_int* y, atomic_int* x) {
  atomic_store_explicit(x, 1, memory_order_relaxed);
  atomic_thread_fence(memory_order_release);
  atomic_store_explicit(y, 1, memory_order_relaxed);
}

void thread1(atomic_int* y, atomic_int* x) {
  atomic_fetch_add_explicit(y, 1, memory_order_relaxed);
  atomic_thread_fence(memory_order_acquire);
  int r0 = atomic_load_explicit(x, memory_order_relaxed);
}

void thread2(atomic_int* y) {
  int r1 = atomic_load_explicit(y, memory_order_relaxed);
}
```

- Incorrectly optimizing this is a rite of passage for compiler engineers: GCC#509632, LLVM#68428, LLVM#72747.

- C++ Memory Model forbids:
  y == 2 && r0 == 1 && r1 == 0
  but replacing fetch_add with reduction introduces this outcome

C++ P3111: Atomic Reduction Operations

Allows programmer to express intent & LLVM to optimize

```
namespace std {

template <class T>
struct atomic {
  void reduce_add(T, memory_order);
};

} // namespace std
```

- Extends C++ atomic and atomic_ref with atomic reductions.
- Allows programs to express intent and implementations to leverage HW.
- Fall-back to “fetch_” add is compliant.

Reduction sequences

Enables combining atomic reduction operations: tree reductions.

```
atomic<float> x = a;

// Allows this…
x.reduce_add(b, relaxed);
x.reduce_add(c, relaxed);

//…to be combined into this:
x.reduce_add(b + c, relaxed);
```

- Enable tree-reductions for floating-point operations.
- Reduction sequence: *maximal contiguous sub-sequence of side effects in the modification order of M, where each operation is an atomic reduction operation.*
- Concurrent atomic reductions are not required to be performed “sequentially” in memory, but can form “tree shapes”.

Weaker Forward Progress guarantees

Atomic Reductions are usable within unsequenced execution

```
for_each(execution::par_unseq, …,
 [](auto) {
  x.fetch_add(0, relaxed); // UB
  x.reduce_add(b, relaxed);// OK
});
```

- C++ parallel algorithms with par_unseq execution policy forbid certain atomic operations to guarantee forward progress.
- Atomic Reductions do not exhibit the issues and can be allowed.

Atomic Reduction Operations

Summary: C++ proposal to enable programs to leverage HW.

C++ atomics

- atomic::reduce_<key>
- atomic::compare_store

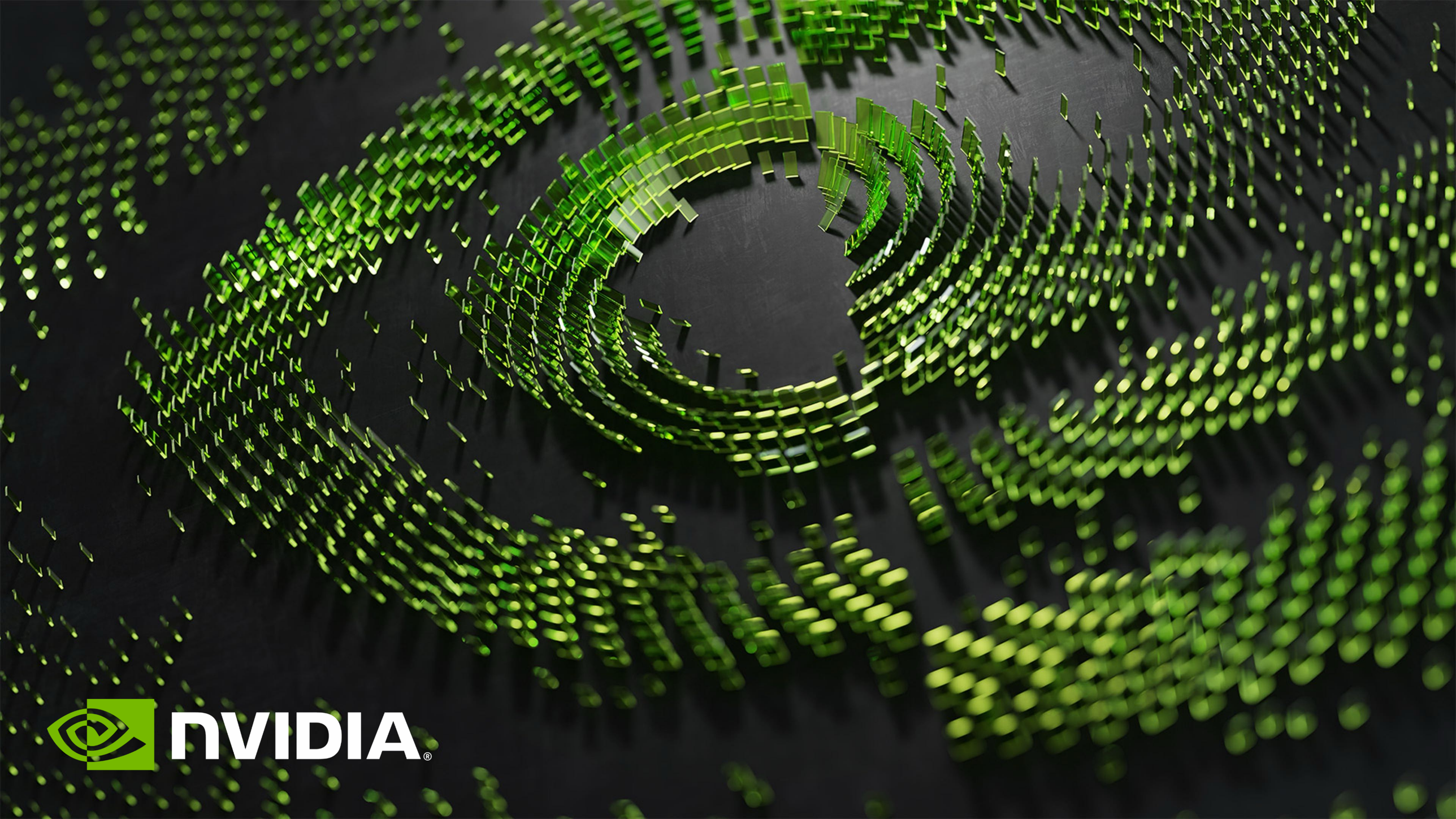
LLVM IR:

- llvm.atomicred…
- llvm.cmpstore…

Backend -> HW:

- Arm: STADD.
- X86: RAO.
- PTX: red.

- LLVM: target-independent intrinsics.
- Floating-Point tree reduction support.
- Forward-Progress SIMD-like schedulers support.


NVIDIA