Lecture 7

- HW 2 out-ish

- Grading HW 1

- First talks Fri
‘Conventional’ vs Atomic Memory Update
Outline

Introduction

Machine Abstractions

C

OpenCL/CUDA

Convergence, Differences in Machine Mapping

Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Introduction

Machine Abstractions

C
OpenCL/CUDA
Convergence, Differences in Machine Mapping
Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Atomic Operations: Compare-and-Swap

```c
#include <stdatomic.h>

Bool atomic_compare_exchange_strong(
    volatile A* obj,
    C* expected, C desired);
```

What does `volatile` mean?
- Any access is considered a side effect

What does this do?
- Check that `*obj` equals `*expected`.
  If so, overwrite `*obj` with `desired`.

How might you use this to implement atomic FP multiplication?
```c
do read prev.
    update (CAS, not successful)
```
Memory Ordering

Why is Memory Ordering a Problem?

- 000 (Phs) reorder ops.
- Compilers

What are the different memory orders and what do they mean?

1. atomicity is unaffected
2. and then

Types:
- seq. consistent: reorder nothing
- acquire: later loads may not reorder across
- release: earlier writes may not
- relaxed: reorder anything

HW: memory forces
Compiler: don’t reorder
Example: A Semaphore With Atomics

```c
#include < nonatomic.h> // mo -> memory_order, a -> atomic
typedef struct { atomic_int v; } naive_sem_t;
void sem_down(naive_sem_t *s)
{
    while (1) {
        while (a_load_explicit(&(s->v), mo_acquire) < 1)
            spinloop_body();
        int tmp = a_fetch_add_explicit(&(s->v), -1, mo_acquire);
        if (tmp >= 1)
            break; // we got the lock
        else // undo our attempt
            a_fetch_add_explicit(&(s->v), 1, mo_relaxed);
    }
}
void sem_up(naive_s_t *s) {
    a_fetch_add_explicit(&(s->v), 1, mo_relaxed);
}

[Cordes '16] — Hardware implementation: how?
C: What is 'order'?

C11 Committee Draft, December ‘10, Sec. 5.1.2.3, “Program execution”:

- (3) _Sequenced before_ is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are _indeterminately sequenced_ when A is sequenced either before or after B, but it is unspecified which. The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the _sequence points_ is given in annex C.)

Q: Where is this definition used (in the standard document)?

_used to order atomic operations_
C: What is ‘order’? (Encore)

C11 Draft, 5.1.2.4 “Multi-threaded executions and data races”:

- All modifications to a particular atomic object M occur in some particular total order, called the modification order of M.
- An evaluation A carries a dependency to an evaluation B if . . .
- An evaluation A is dependency-ordered before an evaluation B if . . .
- An evaluation A inter-thread happens before an evaluation B if . . .
- An evaluation A happens before an evaluation B if . . .

Why is this so subtle?

Keep juicy reordering opt
While establishing just enough off-core order
C: How Much Lying is OK?

C11 Committee Draft, December ‘10, Sec. 5.1.2.3, “Program execution”:

- (1) The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

- (2) Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. […]
C: How Much Lying is OK?

► (4) In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).
► (6) The least requirements on a conforming implementation are:
  ► Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
  ► At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
  ► The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input. This is the observable behavior of the program.
Arrays

Why are arrays the dominant data structure in high-performance code?

- Performance demands regularity (data layout/flow control)
- Linear algebra

Any comments on C’s arrays?

- 1D arrays
- 2D arrays
- No arrays
Arrays vs Abstraction

Arrays-of-Structures or Structures-of-Arrays? What’s the difference? Give an example.

Language aspects of the distinction? Salient example?
Name language mechanisms for SIMD:

- Inline assembly
- Intrinsics
- Vector types
- "pragma loop unroll" or "unroll loop" prefix
- Loading and modifying of scalar variables

Contrast outer-loop vs inner-loop vectorization.

Side q: Would you consider GPUs outer- or inner-loop-vectorizing?  
[Franchetti/Püschel ‘08]