# matrix arithmetic and tuning in ocaml

colin shaw
08.11.2016

1. **introduction**
    a. welcome to computer science club
    b. thank you to revunit for sponsoring
    c. what we talked about last time
        i. multiple layer perceptrons
        ii. used a mix of lists and blas to accomplish math
    d. what are we talking about today
        i. recap of multiple layer perceptron dsl (from last meetup)
        ii. matrix arithmetic
        iii. applicative / functional programming
        iv. implementation examples
            1. functional ocaml
            2. imperative ocaml
            3. c implementation
        v. observe practical performance (matrix-vector multiplication)
            1. functional ocaml
            2. imperative ocaml
            3. lacaml (fortran) reference implementation
        vi. look at assembly for clues about performance
        vii. tools for integrating ocaml with c
    e. what we are not talking about today
        i. how to perform basic matrix arithmetic operations
        ii. matrix-matrix multiplication optimization (strassen's method, etc.)

2. **recap of multiple layer perception dsl**
    a. expressive
    b. not performant
    c. matrix operation was the bottleneck motivating current goals
    d. current goals
        i. learn about matrix arithmetic implementations
        ii. understand ocaml (and list-based functional language) performance issues
        iii. learn techniques to improve performance in ocaml
        iv. create performant, expressive dsl

3. **matrix arithmetic**
    a. two perspectives
        i. general organization for parallel computation
        ii. linear systems and their solutions (specific organization)
    b. diagram and discuss
        i. scalar
        ii. vector
        iii. matrix
    c. organization of matrices
        i. row major
        ii. column major
    d. indexing strategies
        i. computers are turing machines and have one dimensional memory
        ii. array of arrays (list of lists)
            1. first entity is pointers to second
            2. can be useful for sparse matrices
            3. heap could be fragmented
        iii. single array (list)
            1. better cache performance
            2. lower and more consistent memory utilization
            3. must be more careful with implementation
    e. element-wise operations
        i. addition
        ii. subtraction
        iii. hadamard product
    f. non-element-wise operations (variations on multiplication)
        i. scalar multiplication
        ii. vector-vector (producing scalar or matrix)
        iii. matrix-vector multiplication
        iv. matrix-matrix multiplication
    g. computational complexity and limiting factors
        i. blas 1 - $O(n)$ size problem, $O(n)$ computation (memory bandwidth limited)
        ii. blas 2 - $O(n^2)$ size problem, $O(n^2)$ computation (memory bandwidth limited)
        iii. blas 3 - $O(n^2)$ size problem, $O(n^3)$ computation (most algorithmic benefits)

4. **intro to applicative / functional programming**
    a. what differentiates this programming style
        i. my pet peeve - “my imperative language now has ___ functional feature”
        ii. first class functions
        iii. higher order functions
        iv. currying
        v. application of functions
        vi. problem is solved as one large application
        vii. no side effects
        viii. provable computation

b. by example - list.ml
    i. basic list type
    ii. applicative style of programming
    iii. recursive functions
    iv. call stack and tail recursion optimization
        1. draw of recursion of non tail-recursive
        2. illustrate call stack problem
        3. draw out tail-recursive as counterexample
c. parametric polymorphism - adds overhead to execution
d. modules (will see in code review)
    i. code organization
e. module signatures (will see in code review)
    i. means of constraining module interface
    ii. might liken to an object oriented interface
f. functors (will see in the code review)
    i. mapping of module to module
    ii. closest object oriented construct
        1. “this” or “self” pseudo-operator
        2. somewhat akin to a module identity endofunctor
        3. pseudo-operator is more syntax, has very little power
    iii. facilitates constructs like monads
    iv. will see in more detail in the code examples
g. pros and cons of applicative / functional programming
    i. pros
        1. provable execution
        2. no side effects
    ii. cons
        1. slower for some applications - counterexample of binary search tree
        2. higher cognitive barrier

**5. matrix arithmetic implementations**

a. functional
    i. list of lists
    ii. lots of reversing for tail recursion
b. imperative
    i. array of arrays
    ii. straightforward implementation
    iii. scope of boxing becomes larger
c. classic c implementation
    i. similar to imperative ocaml
    ii. more straightforward implementation
    iii. no automatic boxing

6. **observational performance**
    a. definition of problem
        i. timing initialization
        ii. timing matrix operation
    b. run the examples
        i. raspberry pi baseline double precision (92.9 mflops)
        ii. functional ocaml (0.9 mflops)
        iii. imperative ocaml (9.2 mflops)
        iv. lacaml (62.7 mflops)

7. **assembly considerations**
    a. considerations
        i. compiled
        ii. collated and cleaned of alignment meta-data, etc.
    b. code review
        i. functional ocaml
        ii. imperative ocaml
        iii. c
    c. strategies to improve performance
        i. use non-tail recursive form when possible
            1. minimizes reversals
            2. minimizes garbage collection
            3. minimizes function call overhead
        ii. avoid imperative ref values tests
            1. use mutable type values
        iii. use arrays instead of lists
            1. loses functional feel
        iv. write critical sections in c while maintaining problem abstractions in ocaml
            1. same example as before 166 mflops single precision

8. **integrating ocaml with foreign languages**
    a. native headers
        i. fine control
        ii. best native code access method
        iii. best for new development
    b. foreign function interface (ffi)
        i. little to no foreign code
        ii. higher execution cost
        iii. less control
        iv. best for integrating existing code

9. **questions / comments / concerns**