[**Cache oblivious array operations**](https://0fps.net/2013/05/28/cache-oblivious-array-operations/)

Posted on [May 28, 2013](https://0fps.net/2013/05/28/cache-oblivious-array-operations/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

I must confess that secretly the article I wrote [last time](https://0fps.wordpress.com/2013/05/22/implementing-multidimensional-arrays-in-javascript/) (in which I introduced [ndarrays](https://github.com/mikolalysenko/ndarray)) was just a pretext to introduce the stuff that I am going to write about today: which is the [cwise](https://github.com/mikolalysenko/cwise) library for array operations in JavaScript.

Array operations

Briefly, [array operations](http://www.phy.ornl.gov/csep/pl/node15.html) are component-wise operations that can be applied across multiple multidimensional arrays simultaneously.  Array operations can be used to implement basic vector arithmetic, like addition or scaling and are a fundamental tool in linear algebra/numerical computing in general.  Because they are so ubiquitous, many languages have special syntax for array operations as well as routines for specifically optimizing the performance; for example in [MATLAB](https://en.wikipedia.org/wiki/MATLAB) if you prefix an operator with . then executes component-wise.  In the functional programming world, array operations can be implemented as a sequence of [zip](http://stackoverflow.com/questions/1115563/what-is-zip-functional-programming) /[map](http://en.wikipedia.org/wiki/Map_(higher-order_function)) /[reduce](http://en.wikipedia.org/wiki/Fold_(higher-order_function)) higher order functions, while in a language with procedural constructs like C or JavaScript you would probably just use a for-loop like this:

for(i=0; i<shape[0]; ++i) {

for(j=0; j<shape[1]; ++j) {

// ... do operation ...

a[i][j] += b[i][j]

}

}

The goal of this article is to investigate different ways to implement these sorts of array operations across multiple ndarrays of various shapes and strides.

Simple algorithm

The simple for-loop algorithm is a good starting point for implementing operations on ndarrays.  However, in an ndarray performing a lookup at each loop iteration introduces an overhead of O(d) (where d is the dimension  of the ndarray) due to the extra indexing and multiplication required.  To avoid doing this, we can compute the index into the underlying array incrementally.  As a sketch of how this works, consider the following psuedocode:

var a\_ptr = a.offset

, b\_ptr = b.offset

for(var i=0; i<a.shape[0]; ++i) {

for(var j=0; j<a.shape[1]; ++j) {

//... do operation ...

a.data[a\_ptr] += b.data[b\_ptr]

//Advance pointers along axis 1:

a\_ptr += a.stride[1]

b\_ptr += b.stride[1]

}

//Advance pointers along axis 0:

a\_ptr += a.stride[0] - a.stride[1] \* a.shape[1]

b\_ptr += b.stride[0] - b.stride[1] \* a.shape[1]

}

Using this new algorithm, the next index for the arrays can be computed by just adding some constant shift to the pointer instead of performing a series of multiplies.  This is indeed a good thing, and one can easily show that in the conventional [RAM model of computation](http://en.wikipedia.org/wiki/Random-access_machine#Introduction_to_the_model), (which is heavily emphasized in undergraduate CS curricula), this approach is optimal.

But as everyone who has ever tried hard to optimize code for processing arrays knows, this is hardly the full story.  Real computers have [hierarchical memory](http://en.wikipedia.org/wiki/Memory_hierarchy), and execute block IO operations on chunks of cache.  Failing to take this into account when analyzing an algorithm can lead to wildly different (and inaccurate) conclusions about its performance.

Two-level memory and striding

So if we want to come up with a better algorithm for array operations we need a better theory of how our computer works.  And perhaps the simplest extension of the RAM model which does this is the *two-level memory model*.  The basic idea in the two-level model is that you are allowed to operate on memory in contiguous chunks of up to size B words at a time.  This is basically a block IO version of the RAM model, and accurately models many realistic algorithms.  In general, the running time of an algorithm on the RAM model is an upper bound on its running time in the two-level model, and the fastest that we can ever speed up an algorithm in the two level model is by a factor of O(B).

This fact is especially important when we are considering array operations and illustrates the importance of striding in our algorithms.  For example, suppose in the above algorithm that the arrays were contiguously packed in [row major order](http://en.wikipedia.org/wiki/Row-major_order), then each of the memory reads could be executed sequentially and the running time would be O(1 + \frac{\text{shape}[0]\text{shape}[1] }{B}).  On the other hand, if the arrays were stored in column major order and if we looped over them in the same order, then we would get no advantage from blocked IO operations and so the cost of executing the algorithm would be O( \text{shape}[0] \text{shape}[1] ).

To fix this, we should iterate over the indices where the largest strides are in the outer loop and the smallest strides are packed inside.  This makes the most efficient use of each block memory operation, and reduces the total computation time by a factor of O(B) (which is optimal) if the stride orderings are compatibile in each array.

Cache aware algorithm

So the next question is what happens if the strides are not compatible?  For example, suppose that *a* was column major while *b* was row major?  Then in the two level model there is no way to take advantage of block memory operations to speed up our loop, and so it would seem that we are back to RAM model performance.  However, there is an important aspect of hierarchical memory which the two level model neglects: **caching**.

But before we can understand and exploit this new technology, we need some sort of model to describe how it works.  For this purpose, the minimal extension of the two level  memory model is known as the *external memory model*.  Like the two level memory model, we can read and write memory in chunks, but in addition there is also a separate (but small and finite) memory called the *cache* which can store up to M words at a time.  We assume that accessing cache is infinitely fast, and so the time complexity of any algorithm in the external memory model is bounded by the number of reads/writes to the external memory source.

This model may seem slightly absurd since it is possible to execute things like infinite loops for free (as long as they only use the cache), however it turns out to be pretty good accurate for numerical computations and data structures, which are all asymptotically limited by the amount of time they spend moving bits around at the slowest level of the memory hierarchy.

Now it turns out that within the external memory model there is an asymptotically efficient algorithm for array operations that works regardless of their stride.  The first place where this seems to have been written down is in the following paper:

A. Aggarwal, J. Vitter. “[The input/output complexity of sorting and related problems](http://dl.acm.org/citation.cfm?id=48535)” (1987) CACM

The general idea is that we break our arrays down into chunks which have shapes that are on the order of B in each dimension.  Then we iterate over the whole matrix in these chunks and execute the array operation across each chunk independently.  In psuedocode, it works like this:

//Assume shape[0], shape[1] are multiples of B for simplicity

for(var i=0; i<shape[0]; i+=B) {

for(var j=0; j<shape[1]; j+=B) {

var a\_ptr = i \* a.stride[0] + j \* a.stride[1]

var b\_ptr = i \* b.stride[0] + j \* b.stride[1]

for(var k=0; k<B; ++k) {

for(var l=0; l<B; ++l) {

a.data[a\_ptr] += b.data[b\_ptr]

a\_ptr += a.stride[1]

b\_ptr += b.stride[1]

}

a\_ptr += a.stride[0] - a.stride[1] \* B

b\_ptr += b.stride[0] - b.stride[1] \* B

}

}

}

Now if M > B^d, then this approach can be used to speed up any array operation by a factor B – regardless of their stride!  Pretty neat!

Cache oblivious algorithm

The above approach is awesome, but it only works if we know how big B and M are.  However, how can we figure out what these parameters are?  One possible solution is to run lots of experiments and directly optimize those values.  This benchmark-first approach to performance tuning works pretty well for known hardware configurations, like game consoles or iPhone and can produce awesome results.  However it is obviously not very robust, nor is it even feasible in all situations, like web applications for example which need to perform well across many different browsers and devices.

Similarly, we often can’t count on the values for M and B to remain constant throughout the life time of a program.  Contention for disk, RAM and CPU cache can cause the amount of available cache and bandwidth to fluctuate, and so we really shouldn’t assume these things are constant.

Finally, there is also the small problem that computers typically have more than one level of caching, and optimizing array operations for multiple levels of cache would require introducing one extra loop per layer, which can be quite impractical to implement.

Fortunately there is a single elegant model of computation which solves all of these problems at once.  It is known as the cache oblivious memory model, and it was invented by Harald Prokop:

H. Prokop. “[Cache-oblivious algorithms](http://ocw.raf.edu.rs/courses/electrical-engineering-and-computer-science/6-895-theory-of-parallel-systems-sma-5509-fall-2003/readings/cach_oblvs_thsis.pdf)” (1999) MIT Master’s Thesis.

For the amount of problems that it solves, the cache oblivious model of computation is remarkably simple: it is just the external memory model, except we are not allowed to know B and M.  But despite being so trivial, the cache oblivious model has a number of remarkable properties:

1. Algorithms that scale in the cache oblivious model scale across all levels of the memory hierarchy simultaneously.
2. Cache oblivious algorithm “just work” across any type of hardware or virtual machine with no fine tuning.
3. Cache oblivious algorithms automatically throttle with hardware availability and scale performance accordingly.
4. Finally, programs written for the cache oblivious model look just like ordinary RAM model programs!  There is no need to introduce extra notation or operations.

These properties make the cache oblivious model the gold standard for analyzing high performance algorithms, and it should always be our goal to design data structures and algorithms that scale in this model.  However, to achieve these benefits we need to do a bit more work up front when we are designing our algorithms.

For array operations, a simple way to do this is to adapt Prokop’s cache oblivious matrix transpose algorithm (see the above thesis for details).  The way it works is that we keep recursively splitting our multidimensional arrays in half along the largest axis until finally we are left with blocks of unit size.  In psuedo-JavaScript using [ndarrays](https://github.com/mikolalysenko/ndarray), it would look like this:

function recursiveAdd(a, b) {

var rows = a.shape[0]

, cols = a.shape[1]

if(rows === 1 && cols === 1) {

a.data[a.offset] += b.data[b.offset]

} else if(rows > cols) {

rows >>>= 1

recursiveAdd(a.hi(rows, cols), b.hi(rows, cols))

recursiveAdd(a.lo(rows, 0), b.lo(rows, 0))

} else {

cols >>>= 1

recursiveAdd(a.hi(rows, cols), b.hi(rows, cols))

recursiveAdd(a.lo(0, cols), b.lo(0, cols))

}

}

The above code also works regardless of the stride and always uses an optimal O(1+\frac{\text{shape}[0]\text{shape}[1]}{B}) block memory transfers if the cache is at least M \geq B^2.  It is also pretty easy to generalize this idea to multiple arrays with higher dimensions, which again performs optimally assuming that M \geq n B^d where *n*is the number of arguments and *d*is the dimension of the arrays.

Array operations for JavaScript

The above cache oblivious algorithm is clearly what we want to implement, but the code in the previous listing is just a sketch and if implemented naively is not likely to be very fast (at least in JavaScript).  There are quite a few important JS-specific micro-optimizations that we ought to apply before seriously implementing cache-oblivious array operations:

* Slicing the arrays creates a small amount of garbage at each level of recursion, and so we should probably flatten out the objects and keep track of their structures as arguments incrementally to avoid unnecessary memory allocation.
* Using a recursive function call is much slower than manually using a stack to keep track of previous calls,.
* Similarly, we should use the pointer tricks from the simple algorithm to speed up array indexing.
* Also if the strides match, we should use the direct pointer arithmetic version as it has lower overhead
* For small array sizes the overhead of recursion is much larger than the savings due to cache improvements, and so we would want to terminate the subdivision sometime before getting all the way down to unit arrays.  (This is not related to the size of the cache blocks since it is a function of the amount of CPU overhead, not memory bandwidth).

Code!

All of these issues can be pretty tricky to get right, and so it makes sense to try to push all of this complexity into a library.  This is what the [cwise](https://github.com/mikolalysenko/cwise) library does, and you can get it here:

<https://github.com/mikolalysenko/cwise>

That library should work in any JavaScript environment that supports typed arrays, including most current browsers (IE 9+, Firefox, Chrome and Safari).  If you want to try it out in your own projects, you can install it via npm or use [browserify](http://browserify.org/) (or some similar tool) to stick it in your web page.

Macros and dynamic code generation

The basic idea behind cwise is similar to libraries like [FFTW](http://www.fftw.org/) which dynamically generate tuned code based on their inputs (though because it is much easier to do dynamic code generation in JavaScript the interface for cwise is a bit simpler).  The way it works is that it lazily compiles optimized scans for array operations based on the shape and stride of the input arguments.  You can specify the component-wise operations for cwise using ordinary JavaScript which gets parsed and compiled using [esprima](http://esprima.org/)/[falafel](https://github.com/substack/node-falafel) at run time.  This is only done the first time you execute a cwise function, all subsequent calls reuse the same optimized script.

Tricks

There are a lot of useful things that you can do with ndarrays using cwise.  There are plenty of useful recipes on the [github page](https://github.com/mikolalysenko/cwise), but to help get things started here is a quick survey of some stuff you can do:

Vector arithmetic

The most obvious use of cwise is to implement basic vector arithmetic.  You can grab a handful of prebaked ready-to-go operations from the following github project:

<https://github.com/mikolalysenko/ndarray-ops>

As a rule, using these sort of helper methods is not as efficient as unpacking your array operations into a cwise data structure, but on the other hand they can simplify your code and for basic tasks they are often fast enough.

Matrix transpose

A more complicated example of using ndarrays is to perform matrix transpose or cache oblivious reindexing of arrays.  For example, this is pretty easy to do by just changing the stride in the target.  Suppose for example, suppose we want to transpose a square image which is stored in column-major format.  Then we can do this using the following code:

var ops = require("ndarray-ops")

var ndarray = require("ndarray")

function transpose(output, input, size) {

var a = ndarray.ctor(output, [size, size], [size, 1], 0)

, b = ndarray.ctor(input, [size, size], [1, size], 0)

ops.assign(a, b)

}

This is essentially the same thing that Prokop’s cache oblivious matrix transpose algorithm does.

Array copying

Another neat trick we can do with array operations is make copies of arrays in memory.  For example, if we want to fill an array with a repeating pattern, we can use the following snippet:

var ops = require("ndarray-ops")

var ndarray = require("ndarray")

function fillPattern(output, pattern, count) {

var a = ndarray.ctor(output, [count, pattern.length], [pattern.length, 1], 0)

, b = ndarray.ctor(pattern, [count, pattern.length], [0, 1], 0)

ops.assign(a, b)

}

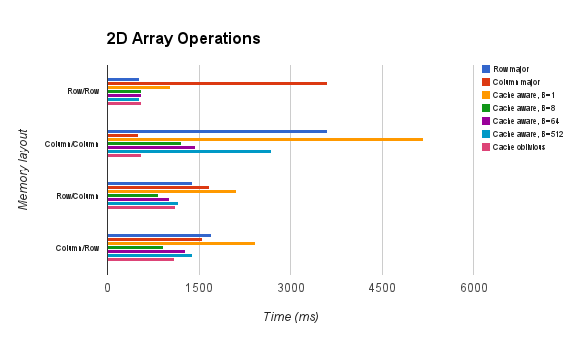
This generalizes to filling 2D images with repeated tiles or for stuff like slicing volume data apart into a texture.

Experiments

So how much difference does all of this make?  To add some numbers to all of these different array operations I made a few quick experiments in node.js to try everything out.  You can check out the results for yourself here on github:

<https://github.com/mikolalysenko/cwise-experiment>

And here are the results summarized in graphical format:

[](https://0fps.files.wordpress.com/2013/05/chart_1-3.png)

Comparison of various algorithms for 2D array operations for different memory layouts.

We can draw the following conclusions from these measurements:

* When possible, sequential iteration is the optimal implementation of array operations.
* For non-compatible strides, cache aware array operations are fastest, but they require knowledge of the underlying architecture (which may not always be available)
* Though cache oblivious array operations are almost never the fastest possible algorithm, they come closest across the widest array of possible inputs.

Conclusion

In summary, we’ve discussed an asymptotically optimal algorithm cache-oblivious array operations across many multidimensional arrays with arbitrary shapes and strides.  We’ve also seen an implementation of these concepts in JavaScript via the [ndarray](https://github.com/mikolalysenko/ndarray) and [cwise](https://github.com/mikolalysenko/cwise) libraries (that should work in node.js and in any browser that supports typed arrays).  These tools enable a wide variety of interesting image processing and linear algebra operations, but a lot more work is still necessary to realize a full suite of numerical linear algebra tools in JavaScript.  Here are a few possible ideas for future projects:

* Tools for basic linear algebra computations (matrix inverse, iterative/direct solvers, eigenvalue problems)
* Data structures for sparse multidimensional arrays
* Libraries for implementing Einstein tensor summation/higher order map-reduce in a style similar to cwise.