Optimizing Code vs Recognizing Patterns with 3D Arrays

James Joseph Balamuta

Intro

As is the case with the majority of posts normally born into existence, there was an interesting problem that arose recently on StackOverflow. Steffen, a scientist at an unnamed weather service, faced an issue with the amount of computational time required by a loop intensive operation in R. Specifically, Steffen needed to be able to sum over different continuous regions of elements within a 3D array structure with dimensions `2500 x 2500 x 50` to acquire the amount of neighbors. The issue is quite common within geography information sciences and/or spatial statistics. What follows next is a modified version of the response that I gave providing additional insight into various solutions.

Problem Statement

Consider an array of matrices that contain only 1 and 0 entries that are spread out over time with dimensions

$$X \times Y \times T$$

. Within each matrix, there are specific regions on the

$$X - Y$$

plane of a fixed time,

$$t$$

, that must be summed over to obtain their neighboring elements. Each region is constrained to a four by four tile given by points inside

$$(x-2, y-2), (x-2, y+2), (x+2, y+2), (x+2, y-2)$$

, where

$$x \in [3, X-2]$$

,

$$y \in [3, Y-2]$$

,

$$X \geq 5$$

, and

$$Y \geq 5$$

.

```
# Matrix Dimensions
xdim <- 20
ydim <- 20

# Number of Time Steps
tdim <- 5

# Generation of 3D Arrays
```

```r
# Initial Neighborhoods
a <- array(0:1, dim = c(xdim, ydim, tdim))

# Result storing matrix
res <- array(0:1, dim = c(xdim, ydim, tdim))

# Calculation loop over time
for (t in 1:tdim) {
    ## Subset by specific rows
    for (x in 3:(xdim-2)) {
        ## Subset by specific columns
        for (y in 3:(ydim-2)) {
            ## Sum over each region within a time point
            res[x,y,t] <- sum(a[(x-2):(x+2), (y-2):(y+2), t])
        }
    }
}
```

Sample Result:

Without a loss of generality, I've opted to downscale the problem to avoid a considerable amount of output while displaying a sample result. Thus, the sample result presented next is under the dimensions:

$$6 \times 6 \times 2$$

. Therefore, the results of the neighboring clusters are:

```
, , 1

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    1    1    1    1    1    1
[3,]    0    0   10   10    0    0
[4,]    1    1   15   15    1    1
[5,]    0    0    0    0    0    0
[6,]    1    1    1    1    1    1

, , 2

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    1    1    1    1    1    1
[3,]    0    0   10   10    0    0
[4,]    1    1   15   15    1    1
[5,]    0    0    0    0    0    0
[6,]    1    1    1    1    1    1
```

Note: Both time points, `t <- 1` and `t <- 2`, are the same!!! More on this interesting pattern later...

Possible Solutions

There are many ways to approach a problem like this. The first is to try to optimize the initial code base within R. Secondly, and one of the primary reasons for this article is, one can port over the computational

portion of the code and perhaps add some parallelization component with the hope that such a setup would decrease the amount of time required to perform said computations. Thirdly, one can try to understand the underlying structure of the arrays by searching for patterns that can be exploited to create a cache of values which would be able to be reused instead of individually computed for each time point.

Thus, there are really three different components within this post that will be addressed:

1. Optimizing R code directly in hopes of obtaining a speedup in brute forcing the problem;
2. Porting over the R code into C++ and parallelizing the computation using more brute force;
3. Trying to determine different patterns in the data and exploit their weakness

Optimizing within R

The initial problem statement has something that all users of `R` dread: a loop. However, it isn't just one loop, it's 3! As is known, one of the key downsides to `R` is looping. This problem has a lot of coverage — my favorite being the straight, curly, or compiled as it sheds light on R's parser — and is primarily one of the key reasons why `Rcpp` is favored in loop heavy problems.

There are a few ways we can aim to optimize just the R code:

1. Cache subsets within the loop.
2. Parallelize the time computation stage.

Original

To show differences between functions, I'll opt to declare the base computational loop given above as:

```
cube_r <- function(a, res, xdim, ydim, tdim){
    for (t in 1:tdim) {
        for (x in 3:(xdim-2)) {
            for (y in 3:(ydim-2)) {
                res[x,y,t] <- sum(a[(x-2):(x+2), (y-2):(y+2), t])
            }
        }
    }
    res
}
```

Cached R

The first order of business is to implement a cached value schema so that we avoid subsetting the different elements considerably.

```
cube_r_cache <- function(a, res, xdim, ydim, tdim){
    for (t in 1:tdim) {
        temp_time <- a[,,t]
        for (x in 3:(xdim-2)) {
            temp_row <- temp_time[(x-2):(x+2),]
            for (y in 3:(ydim-2)) {
                res[x,y,t] <- sum(temp_row[,(y-2):(y+2)])
            }
        }
    }
    res
}
```

Parallelized R

Next up, let's implement a way to parallelize the computation over time using R's built in `parallel` package.

```
library(parallel)

cube_r_parallel <- function(a, xdim, ydim, tdim, ncores){
    ## Create a cluster
    cl <- makeCluster(ncores)

    ## Create a time-based computation loop
    computation_loop <- function(X, a, xdim, ydim, tdim) {
        temp_time <- a[,,X]
        res <- temp_time
        for (x in 3:(xdim-2)) {
            temp_row <- temp_time[(x-2):(x+2),]
            for (y in 3:(ydim-2)) {
                res[x,y] <- sum(temp_row[,(y-2):(y+2)])
            }
        }
        res
    }

    ## Obtain the results
    r_parallel <- parSapply(cl = cl,
                            X = seq_len(tdim),
                            FUN = computation_loop,
                            a = a, xdim = xdim, ydim = ydim, tdim = tdim,
                            simplify = "array")

    ## Kill the cluster
    stopCluster(cl)

    ## Return
    r_parallel
}
```

Validating Function Output

After modifying a function's initial behavior, it is a good idea to check whether or not the new function obtains the same values. If not, chances are there was a bug that crept into the function after the change.

```
r_original  <- cube_r(a, res, xdim, ydim, tdim)
r_cached    <- cube_r_cache(a, res, xdim, ydim, tdim)
r_parallel1 <- cube_r_parallel(a, xdim, ydim, tdim, ncores = 1)

all.equal(r_original, r_cached)
```

```
## [1] TRUE
```

```
all.equal(r_original, r_parallel1)
```

```
## [1] TRUE
```

Thankfully, all the modifications yielded no numerical change from the original.

Benchmarking

As this **is** `Rcpp`, benchmarks are king. Here I've opted to create a benchmark of the different functions after trying to optimize them within R.

```
library("microbenchmark")
rtimings <- microbenchmark(r_orig   = cube_r(a, res, xdim, ydim, tdim),
                           r_cached = cube_r_cache(a, res, xdim, ydim, tdim),
                           r_cores1 = cube_r_parallel(a, xdim, ydim, tdim, ncores = 1),
                           r_cores2 = cube_r_parallel(a, xdim, ydim, tdim, ncores = 2),
                           r_cores3 = cube_r_parallel(a, xdim, ydim, tdim, ncores = 3),
                           r_cores4 = cube_r_parallel(a, xdim, ydim, tdim, ncores = 4))
```

Output:

```
Unit: milliseconds
     expr        min         lq       mean     median         uq        max neval
   r_orig   1.592999   1.842691   2.809963   2.161219   2.697586  45.223308   100
 r_cached   1.137502   1.360473   1.650674   1.504519   1.759342   7.605762   100
 r_cores1 164.277702 167.492365 173.434411 170.753039 176.855520 202.233596   100
 r_cores2 296.510341 304.151671 313.392266 309.457475 319.406546 371.616252   100
 r_cores3 443.118539 453.321725 466.498576 463.555702 475.986821 524.133514   100
 r_cores4 591.027044 607.331826 618.288022 615.347604 627.338268 670.718708   100
```

Not surprisingly, the cached subset function performed better than the original R function. On the other hand, what was particularly surpising was that the parallelization option within R took a considerably longer amount of time as additional cores were added. This was partly due to the fact that the `a` array had to be replicated out across the different `ncores` number of `R` processes spawned. In addition, there was also a time lag between spawning the processes and winding them down. This may prove to be a fatal flaw of the construction of `cube_r_parallel` as a normal user may wish to make repeated calls within the same parallelized session. However, I digress as I feel like I've spent too much time describing ways to optimize the `R` code.

Porting into C++

To keep in the spirit of optimizing a generalized solution, we can quickly port over the R cached version of the computational loop to `Armadillo`, C++ Linear Algebra Library, and use `OpenMP` with `RcppArmadillo`.

Why not use just `Rcpp` though? The rationale for using `RcppArmadillo` over `Rcpp` is principally because of the native support for multidimensional arrays via arma::cube. In addition, it's a bit painful in the current verison of `Rcpp` to work with an array. Hence, I would rather face the cost of copying an object from `SEXP` to `arma` and back again than mess around with this.

Before we get started, it is very important to provide protection for systems that still lack `OpenMP` by default in R (**cough** OS X **cough**). Though, by changing how R's `~/.R/Makevars` compiler flag is set, `OpenMP` can be used within `R`. Discussion of this approach is left to http://thecoatlessprofessor.com/programming/openmp-in-r-on-os-x/. Now, there still does exist the real need to protect users of the default R `~/.R/Makevars` configuration by guarding the header inclusion of `OpenMP` in the `cpp` code using preprocessor directives:

```
// Protect against compilers without OpenMP
#ifdef _OPENMP
```

```
#include <omp.h>
#endif
```

Given the above code, we have effectively provided protection from the compiler throwing an error due to `OpenMP` not being available on the system. **Note:** In the event that the system does not have `OpenMP`, the process will be executed serially just like always.

With this being said, let's look at the C++ port of the R function:

```
#include <RcppArmadillo.h>

// Correctly setup the build environment
// [[Rcpp::depends(RcppArmadillo)]]

// Add a flag to enable OpenMP at compile time
// [[Rcpp::plugins(openmp)]]

// Protect against compilers without OpenMP
#ifdef _OPENMP
#include <omp.h>
#endif

// Parallelized C++ equivalent
// [[Rcpp::export]]
arma::cube cube_cpp_parallel(arma::cube a, arma::cube res, int ncores = 1) {

    // Extract the different dimensions

    // Normal Matrix dimensions
    unsigned int xdim = res.n_rows;

    unsigned int ydim = res.n_cols;

    // Depth of Array
    unsigned int tdim = res.n_slices;

    // Added an omp pragma directive to parallelize the loop with ncores
    #pragma omp parallel for num_threads(ncores)
    for (unsigned int t = 0; t < tdim; t++) { // Note: Array Index in C++ starts at 0 and goes to N - 1

        // Pop time `t` from `a` e.g. `a[,,t]`
        arma::mat temp_mat = a.slice(t);

        // Begin region selection
        for (unsigned int x = 2; x < xdim-2; x++) {

            // Subset the rows
            arma::mat temp_row_sub = temp_mat.rows(x-2, x+2);

            // Iterate over the columns with unit accumulative sum
            for (unsigned int y = 2; y <  ydim-2; y++) {
                res(x,y,t) = accu(temp_row_sub.cols(y-2,y+2));
            }
        }
    }
```

```
    return res;
}
```

A few things to note here:

1. `a`, `res`, `xdim`,`ydim` and `ncores` are **shared** across processes.
2. `t` is unique to each process.
3. We protect users that cannot use `OpenMP`!

To verify that this is an equivalent function, we opt to check the object equality:

```
cpp_parallel <- cube_cpp_parallel(a, res, ncores = 2)

all.equal(cpp_parallel, r_original)
```

```
## [1] TRUE
```

Timings

Just as before, let's check the benchmarks to see how well we did:

```
port_timings <- microbenchmark(r_orig    = cube_r(a, res, xdim, ydim, tdim),
                               r_cached  = cube_r_cache(a, res, xdim, ydim, tdim),
                               cpp_core1 = cube_cpp_parallel(a, res, 1),
                               cpp_core2 = cube_cpp_parallel(a, res, 2),
                               cpp_core3 = cube_cpp_parallel(a, res, 3),
                               cpp_core4 = cube_cpp_parallel(a, res, 4))
```

Output:

```
Unit: microseconds
      expr      min        lq       mean    median        uq      max neval
    r_orig 1523.984 1717.9330 2028.63662 1774.2060 1906.9265 5955.428   100
  r_cached 1129.008 1252.3500 1430.79695 1317.2945 1417.8080 3044.783   100
 cpp_core1   40.348   45.1255   52.57591   48.8420   53.6200  119.980   100
 cpp_core2   47.073   60.6980   87.24594   75.2090   92.5515  249.869   100
 cpp_core3   45.656   71.6700  136.74181   93.0820  157.8490 1371.444   100
 cpp_core4   56.982   92.3740  210.72895  186.5175  259.6020  705.719   100
```

Wow! The C++ version of the parallelization really did wonders when compared to the previous R implementation of parallelization. The speed ups when compared to the R implementations are about 44x vs. original and 34x vs. the optimized R loop (using 3 cores). Note, with the addition of the 4th core, the parallelization option performs poorly as the system running the benchmarks only has four cores. Thus, one of the cores is trying to keep up with the parallelization while also having to work on operating system tasks and so on.

Now, this isn't to say that there is no cap to parallelization benefits given infinite amounts of processing power. In fact, there are two laws that govern general speedups from parallelization: Amdahl's Law (Fixed Problem Size) and Gustafson's Law (Scaled Speedup). The details are better left for another time on this matter.

Detecting Patterns

Previously, we made the assumption that the structure of the data within computation had no pattern. Thus, we opted to create a generalized algorithm to effectively compute each value. Within this section, we remove

the assumption about no pattern being present. In this case, we opt to create a personalized solution to the problem at hand.

As a first step, notice how the array is constructed in this case with: `array(0:1, dims)`. There seems to be some sort of pattern depending on the `xdim`, `ydim`, and `tdim` of the distribution of 1s and 0s. If we can recognize the pattern in advance, the amount of computation required decreases. **However, this may also impact the ability to *generalize* the algorithm to other cases outside the problem statement.** Thus, the reason for this part of the post being at the terminal part of this article.

After some trial and error using different dimensions, the different patterns become recognizable and reproducible. Most notably, we have three different cases:

- Case 1: If `xdim` is even, then only the rows of a matrix alternate with rows containing **all** 1s or 0s.
- Case 2: If `xdim` is odd and `ydim` is even, then only the rows alternate of a matrix alternate with rows containing a **combination of both** 1 or 0.
- Case 3: If `xdim` is odd and `ydim` is odd, then rows alternate as well as the matrices alternate with a **combination of both** 1 or 0.

Examples of Pattern Cases

Let's see the cases described previously in action to observe the patterns. Please note that the dimensions of the example case arrays are small and would likely yield issues within the `for` loop given above due to the indices being negative or zero triggering an out-of-bounds error.

Case 1:

```
xdim <- 2
ydim <- 3
tdim <- 2
a <- array(0:1, dim = c(xdim, ydim, tdim))
```

Output:

```
a
```

Case 2:

```
xdim <- 3
ydim <- 4
tdim <- 2
a <- array(0:1, dim = c(xdim, ydim, tdim))
```

Output:

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    0    1    0    1
[2,]    1    0    1    0
[3,]    0    1    0    1

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    1    0    1
[2,]    1    0    1    0
[3,]    0    1    0    1
```

Case 3:

```
xdim <- 3
ydim <- 3
tdim <- 3
a <- array(0:1, dim = c(xdim, ydim, tdim))
```

Output:

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    0    1    0    1
[2,]    1    0    1    0
[3,]    0    1    0    1

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    1    0    1
[2,]    1    0    1    0
[3,]    0    1    0    1
```

Pattern Hacking

Based on the above discussion, we opt to make a bit of code that exploits this unique pattern. The language that we can write this code in is either R or C++. Though, due to the nature of this website, we opt to proceed in the later. Nevertheless, as an exercise, feel free to backport this code into R.

Having acknowledged that, we opt to start off trying to create code that can fill a matrix with either an `even` or an `odd` column vector. e.g.

Odd Vector

The *odd* vector is defined as having the initial value being given by `1` instead of `0`. This is used in heavily in both **Case 2** and **Case 3**.

```
     [,1]
[1,]    1
[2,]    0
[3,]    1
[4,]    0
[5,]    1
```

Even

In comparison to the *odd* vector, the *even* vector starts at 0. This is used principally in **Case 1** and then in **Case 2** and **Case 3** to alternate columns.

```
     [,1]
[1,]    0
[2,]    1
[3,]    0
[4,]    1
[5,]    0
```

Creating Alternating Vectors

To obtain such an alternating vector that switches between two values, we opt to create a vector using the modulus operator while iterating through element positions in an

$$N$$

length vector. Specifically, we opt to use the fact that when `i` is an even number `i % 2` must be `0` and when `i` is odd `i % 2` gives `1`. Thefore, we are able to create an alternating vector that switches between two different values with the following code:

```
#include <RcppArmadillo.h>

// Correctly setup the build environment
// [[Rcpp::depends(RcppArmadillo)]]

// ------- Make Alternating Column Vectors
// [[Rcpp::export]]
// Creates a vector with initial value 1
arma::vec odd_vec(unsigned int xdim) {

    // make a temporary vector to create alternating 0-1 effect by row.
    arma::vec temp_vec(xdim);

    // Alternating vector (anyone have a better solution? )
    for (unsigned int i = 0; i < xdim; i++) {
        temp_vec(i) = (i % 2 ? 0 : 1); // Ternary operator in C++, e.g. if(TRUE){1}else{0}
    }

    return temp_vec;
}

// Creates a vector with initial value 0
// [[Rcpp::export]]
arma::vec even_vec(unsigned int xdim){

    // make a temporary vector to create alternating 0-1 effect by row.
    arma::vec temp_vec(xdim);

    // Alternating vector (anyone have a better solution? )
    for (unsigned int i = 0; i < xdim; i++) {
        temp_vec(i) = (i % 2 ? 1 : 0); // changed
    }

    return temp_vec;
}
```

Creating the three cases of matrix

With our ability to now generate **odd** and **even** vectors by column, we now need to figure out how to create the matrices described in each case. As mentioned above, there are three cases of matrix given as:

1. The even,
2. The bad odd,
3. And the ugly odd.

Using a similar technique to obtain the alternating vector, we obtain the three cases of matrices:

```
// --- Handle the different matrix cases

// Case 1: xdim is even
// [[Rcpp::export]]
arma::mat make_even_matrix_case1(unsigned int xdim, unsigned int ydim) {

    arma::mat temp_mat(xdim,ydim);

    temp_mat.each_col() = even_vec(xdim);

    return temp_mat;
}

// Case 2: xdim is odd and ydim is odd
// [[Rcpp::export]]
arma::mat make_odd_matrix_case2(unsigned int xdim, unsigned int ydim){

    arma::mat temp_mat(xdim,ydim);

    // Cache values
    arma::vec e_vec = even_vec(xdim);
    arma::vec o_vec = odd_vec(xdim);

    // Alternating column
    for (unsigned int i = 0; i < ydim; i++) {
        temp_mat.col(i) = (i % 2 ? e_vec : o_vec);
    }

    return temp_mat;
}

// Case 3: xdim is odd and ydim is even
// [[Rcpp::export]]
arma::mat make_odd_matrix_case3(unsigned int xdim, unsigned int ydim){

    arma::mat temp_mat(xdim,ydim);

    // Cache values
    arma::vec e_vec = even_vec(xdim);
    arma::vec o_vec = odd_vec(xdim);

    // Alternating column
    for (unsigned int i = 0; i < ydim; i++) {
        temp_mat.col(i) = (i % 2 ? o_vec : e_vec); // slight change
    }

    return temp_mat;
}
```

Calculation Engine

Next, we need to create a computational loop to subset the appropriate continuous areas of the matrix to

figure out the amount of neighbors. In comparison to the problem statement, note that this loop is without the `t` as we no longer need to repeat calculations within this approach. Instead, we only need to compute the values once and then cache the result before duplicating it across the 3D array (e.g. `arma::cube`).

```
// --- Calculation engine

// [[Rcpp::export]]
arma::mat calc_matrix(arma::mat temp_mat) {

    // Obtain matrix dimensions
    unsigned int xdim = temp_mat.n_rows;

    unsigned int ydim = temp_mat.n_cols;

    // Create the 2D result matrix with temp_mat initial values
    arma::mat res = temp_mat;

    // Subset the rows
    for (unsigned int x = 2; x < xdim-2; x++){ // Note: Index Shift to C++!

        arma::mat temp_row_sub = temp_mat.rows(x-2, x+2);

        // Iterate over the columns with unit accumulative sum
        for (unsigned int y = 2; y <  ydim-2; y++){
            res(x,y) = accu(temp_row_sub.cols(y-2,y+2));
        }
    }

    return res;
}
```

Call Main Function

Whew, that was a lot of work. But, by approaching the problem this way, we have:

1. Created reusable code snippets.
2. Decreased the size of the main call function.
3. Improved clarity of each operation.

Now, the we are ready to write the glue that combines all the different components. As a result, we will obtain the desired neighbor information.

```
// --- Main Engine

// Create the desired cube information
// [[Rcpp::export]]
arma::cube dim_to_cube(unsigned int xdim = 4, unsigned int ydim = 4, unsigned int tdim = 3) {

    // Initialize values in A
    arma::cube res(xdim,ydim,tdim);

    // Case 1
    if (xdim % 2 == 0) {
        res.each_slice() = calc_matrix(make_even_matrix_case1(xdim, ydim));
    } else { // Either Case 2 or Case 3
```

```
        if (ydim % 2 == 0) { // Case 3
            res.each_slice() = calc_matrix(make_odd_matrix_case3(xdim, ydim));
        } else { // Case 2
            arma::mat first_odd_mat = calc_matrix(make_odd_matrix_case2(xdim, ydim));
            arma::mat sec_odd_mat = calc_matrix(make_odd_matrix_case3(xdim, ydim));

            for (unsigned int t = 0; t < tdim; t++) {
                res.slice(t) = (t % 2 ? first_odd_mat  : sec_odd_mat);
            }
        }
    }
    return res;
}
```

Pieced together

When put together, the code forms:

```
#include <RcppArmadillo.h>

// Correctly setup the build environment
// [[Rcpp::depends(RcppArmadillo)]]

// ------- Make Alternating Column Vectors
// [[Rcpp::export]]
// Creates a vector with initial value 1
arma::vec odd_vec(unsigned int xdim) {

    // make a temporary vector to create alternating 0-1 effect by row.
    arma::vec temp_vec(xdim);

    // Alternating vector (anyone have a better solution? )
    for (unsigned int i = 0; i < xdim; i++) {
        temp_vec(i) = (i % 2 ? 0 : 1); // Ternary operator in C++, e.g. if(TRUE){1}else{0}
    }

    return temp_vec;
}

// Creates a vector with initial value 0
// [[Rcpp::export]]
arma::vec even_vec(unsigned int xdim){

    // make a temporary vector to create alternating 0-1 effect by row.
    arma::vec temp_vec(xdim);

    // Alternating vector (anyone have a better solution? )
    for (unsigned int i = 0; i < xdim; i++) {
        temp_vec(i) = (i % 2 ? 1 : 0); // changed
    }

    return temp_vec;
}
```

```
// --- Handle the different matrix cases

// Case 1: xdim is even
// [[Rcpp::export]]
arma::mat make_even_matrix_case1(unsigned int xdim, unsigned int ydim) {

    arma::mat temp_mat(xdim,ydim);

    temp_mat.each_col() = even_vec(xdim);

    return temp_mat;
}

// Case 2: xdim is odd and ydim is odd
// [[Rcpp::export]]
arma::mat make_odd_matrix_case2(unsigned int xdim, unsigned int ydim){

    arma::mat temp_mat(xdim,ydim);

    // Cache values
    arma::vec e_vec = even_vec(xdim);
    arma::vec o_vec = odd_vec(xdim);

    // Alternating column
    for (unsigned int i = 0; i < ydim; i++) {
        temp_mat.col(i) = (i % 2 ? e_vec : o_vec);
    }

    return temp_mat;
}

// Case 3: xdim is odd and ydim is even
// [[Rcpp::export]]
arma::mat make_odd_matrix_case3(unsigned int xdim, unsigned int ydim){

    arma::mat temp_mat(xdim,ydim);

    // Cache values
    arma::vec e_vec = even_vec(xdim);
    arma::vec o_vec = odd_vec(xdim);

    // Alternating column
    for (unsigned int i = 0; i < ydim; i++) {
        temp_mat.col(i) = (i % 2 ? o_vec : e_vec); // slight change
    }

    return temp_mat;
}

// --- Calculation engine

// [[Rcpp::export]]
arma::mat calc_matrix(arma::mat temp_mat) {
```

```
    // Obtain matrix dimensions
    unsigned int xdim = temp_mat.n_rows;

    unsigned int ydim = temp_mat.n_cols;

    // Create the 2D result matrix with temp_mat initial values
    arma::mat res = temp_mat;

    // Subset the rows
    for (unsigned int x = 2; x < xdim-2; x++){ // Note: Index Shift to C++!

        arma::mat temp_row_sub = temp_mat.rows(x-2, x+2);

        // Iterate over the columns with unit accumulative sum
        for (unsigned int y = 2; y <  ydim-2; y++){
            res(x,y) = accu(temp_row_sub.cols(y-2,y+2));
        }
    }

    return res;
}

// --- Main Engine

// Create the desired cube information
// [[Rcpp::export]]
arma::cube dim_to_cube(unsigned int xdim = 4, unsigned int ydim = 4, unsigned int tdim = 3) {

    // Initialize values in A
    arma::cube res(xdim,ydim,tdim);

    // Case 1
    if (xdim % 2 == 0) {
        res.each_slice() = calc_matrix(make_even_matrix_case1(xdim, ydim));
    } else { // Either Case 2 or Case 3
        if (ydim % 2 == 0) { // Case 3
            res.each_slice() = calc_matrix(make_odd_matrix_case3(xdim, ydim));
        } else { // Case 2
            arma::mat first_odd_mat = calc_matrix(make_odd_matrix_case2(xdim, ydim));
            arma::mat sec_odd_mat = calc_matrix(make_odd_matrix_case3(xdim, ydim));

            for (unsigned int t = 0; t < tdim; t++) {
                res.slice(t) = (t % 2 ? first_odd_mat  : sec_odd_mat);
            }
        }
    }
    return res;
}
```

Verification of Results

To verify, let's quickly create similar cases and test them against the original R function:

Case 1:

```
xdim <- 6; ydim <- 5; tdim <- 3
a <- array(0:1,dim=c(xdim, ydim, tdim)); res <- a

all.equal(dim_to_cube(xdim, ydim, tdim), cube_r(a, res, xdim, ydim, tdim))
```

```
[1] TRUE
```

Case 2:

```
xdim <- 7; ydim <- 6; tdim <- 3
a <- array(0:1,dim=c(xdim, ydim, tdim)); res <- a

all.equal(dim_to_cube(xdim, ydim, tdim), cube_r(a, res, xdim, ydim, tdim))
```

```
[1] TRUE
```

Case 3:

```
xdim <- 7; ydim <- 7; tdim <- 3
a <- array(0:1,dim=c(xdim, ydim, tdim)); res <- a

all.equal(dim_to_cube(xdim, ydim, tdim), cube_r(a, res, xdim, ydim, tdim))
```

```
[1] TRUE
```

Closing Time - You don't have to go home but you can't stay here.

With all of these different methods now thoroughly described, let's do one last benchmark to figure out the best of the best.

```
xdim <- 20
ydim <- 20
tdim <- 5
a <- array(0:1,dim=c(xdim, ydim, tdim))
res <- a

end_bench <- microbenchmark::microbenchmark(r_orig    = cube_r(a, res, xdim, ydim, tdim),
                                            r_cached  = cube_r_cache(a, res, xdim, ydim, tdim),
                                            cpp_core1 = cube_cpp_parallel(a, res, 1),
                                            cpp_core2 = cube_cpp_parallel(a, res, 2),
                                            cpp_core3 = cube_cpp_parallel(a, res, 3),
                                            cpp_cpat  = dim_to_cube(xdim, ydim, tdim))
```

Output:

```
Unit: microseconds
      expr      min        lq       mean    median        uq      max neval
    r_orig 1465.233 1519.5600 1677.40196 1586.8055 1728.5500 2873.839   100
  r_cached 1101.048 1123.6995 1341.56288 1161.9230 1273.5850 6647.343   100
 cpp_core1   39.640   43.8870   47.26357   45.3030   48.1345   85.650   100
 cpp_core2   46.010   61.5835   86.12055   69.9005   84.5885 1327.204   100
 cpp_core3   47.781   67.4230  168.84953  161.2115  232.5270 1276.947   100
  cpp_cpat    8.141   10.9725   14.50466   13.0960   16.2810   41.056   100
```

As can be seen, the customized approach based on the data's pattern provided the fastest speed up. Though, by customizing to the pattern, we lost the ability to generalize the solution without being exposed to new cases. Meanwhile, the code port into C++ yielded much better results than the optimized R version. Both of these pieces of code were highly general to summing over specific portions of an array. Lastly, the parallelization within R was simply too time consuming for a one-off computation.