CS 267 Final Project

Kolen Cheung Katherine Oosterbaan

May 9, 2017

Abstract

We proposed a strategy to speedup the pipeline of POLARBEAR, a group that measures Comic Microwave Background Radiation (CMB) polarization, using Cython such that SIMD vectorization and OpenMP parallelization can be used relatively easily. 3 example functions are focused in this study that involves various different properties and hence different kinds of optimizations required and has different benchmark behaviors.
In the end, we found that the result is generally positive, with a projected speedup of ~ 6 times on Cori's Haswell nodes and ~ 8 times on Cori's KNL nodes. More investigation is needed however. More details in the report.

Contents

1 Introduction

1.1 The Physics

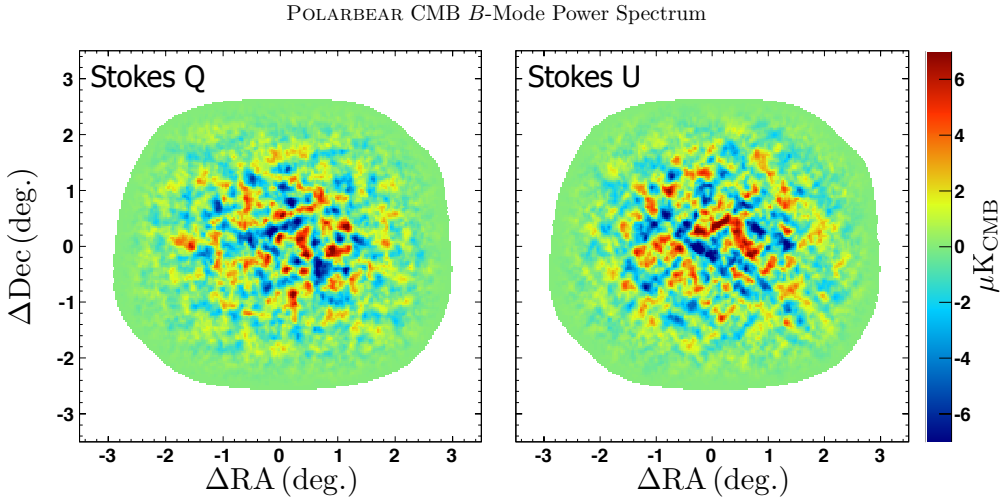
Physicists always push the boundary of our understanding of the most fundamental aspects of the Universe. Some of the fundamental questions we can ask are what constitute the Universe; how gravity plays a role in quantum Physics; why neutrino mass are non-zero, how much mass do they have, and how many numbers of them; and if the current understanding of the Universe through Λ-CDM model is correct.

Many of such questions can only be answered when we probed at higher and higher energy scales. For example, the highest energy scale we can achieve artificially in the state-of-the-art LHC is about ~ 10 TeV, or 10×10^{13} eV. But we can do only so much experimentally because of the limit of the size of the equipment we can build, and it is unlikely for the foreseeable future to create energy scale as high as the GUT scale at ~ 10×10^{16} GeV, or 10×10^{25} eV, which will be important for Quantum Gravity.

So instead of relying on human-built machine, one can measure the primordial signals created by the Universe itself. And the oldest possible such signal that is observable is the Cosmic Microwave Background (CMB) Radiation. It is the first light of the Universe when it was ~ 400000 years young, and everything happened between now and then are imprinted in this signal. Some of the information we can extracts includes gravitational wave at GUT scale (by B-mode analysis on the CMB), dark matter, neutrino mass, falsification of Λ-CDM model, etc.

POLARBEAR is one of the pioneer group on the measurement of CMB polarization in University of California, Berkeley. One of the major result by POLARBEAR in 2014 is(Collaboration et al. 2014):

> the hypothesis of no B-mode polarization power from gravitational lensing is rejected at 97.2% confidence



1.2 The Pipeline

In order to measure the CMB, a telescope is scanning the sky with ~ 100 sensors, each taking a time stream data. An output is created per hour-long observation, called Constant Elevation Scan (CES). There are about ~ 14 such observation per days, and after about 2 years of observation, we accumulate about ~ 10,000 CES.

These data requires a lot of cleanup, partially due to the imperfection of the equipments, and partially due to the uncontrollable factors like weather and atmospheric conditions. In the end, 4237 such inputs are used for the analysis.

Since the correlation time scale is ~ 20 min, shorter than the observation time scale, and we can treat each observation independently. Hence, it is a perfectly (embarrassingly) parallel problem.

In the current implementation of AnalysisBackend, the code is mainly written in Python with some occasional C/C++ modules, and relies heavily on external libraries e.g. Numpy. All code is written in serial, and in the end a trivial MPI process is run to start these individual independent processes.[1]

However, such a perfect parallel treatment is not without its problems:

1. Cori's Haswell nodes has only 128 GB RAM, this limits the number of concurrent in-node processes to about 16, which is significantly less than the 32 number of codes available.
2. Due to the MASTER algorithm (TODO: cite), in the past CMB analysis is mainly $O(n^3)$ which enjoys high computational intensity. The MASTER algorithm improve the scaling to $O(n)$, but this decreases the computational intensity and therefore the pipeline is highly likely to be IO bounded. It also means that it potentially will be speeded up by hyper-threading.
3. Ideally, we would want to run the pipeline using Cori's Knights Landing (KNL) nodes. This means that it is going to be slightly more RAM-limited, and have much more cores and hyper-threading idling. Currently, KNL can afford at most 272 threads to run concurrently, much bigger than the current 16 processes per node we are using.
4. It is worth noting that SIMD vectorization is also important, and is currently unexplored in AnalysisBackend. Haswell CPU has 4-wide and KNL has 8-wide Fused multiply–add (FMA).

It is also worth noting however that some form of SIMD vectorization and OpenMP parallelism are "built-in" from the libraries used. e.g. Numpy is heavily relied upon throughout the pipeline, and currently the Intel Distribution of Python used on NERSC has been compiled with optimization in SIMD and OpenMP support. Many libraries that are shipped with Intel's distribution has these optimizations, including Numpy.

2 Optimization and Parallelization

From the above limitation, we laid out the following requirements:

1. In order for the pipeline to be ready for KNL, SIMD vectorization and OpenMP parallelization is necessary.
2. (non-trivial) MPI[2] is not necessary however, because each CES is still independent, and in the foreseeable future the RAM needed for a single CES will not surpass the RAM available in 1 node on Cori.
3. Written in Python. This has been mostly true in AnalysisBackend. The choice of Python can be attributed to historical reasons, but also because of maintainability (e.g. in writing, packaging, distributing, testing) and readability.

2.1 The Proposal

Our proposal is then to write every module in Cython.

Cython as a language is a superset of Python, and as a compiler is a transpiler to C/C++. It supports both SIMD vectorization and OpenMP parallelism.

Cython is not without its limitation. Directive pragma and Intel intrinsics are not supported, SIMD vectorization relies on the C/C++ compilers' optimization, and obtaining the vectorization report is convoluted. OpenMP support is limited, e.g. `#pragma omp for` can be used in Cython by `prange`, reduction is implied by in-place operators, and as a result other kinds of reduction like `max` are not supported.

Fortunately, these limitations will not be important to POLARBEAR's analysis. As shown in the demonstration below. One of the reason is that there are a lot of inherit parallelism in the pipeline. For example, each channel from the data are usually independently processed, which is a trivial use case of `prange`.

3 Results, Benchmarks, and Discussions

As a proposal on parallelizing the whole pipeline, we focuses on 3 functions in the pipeline to demonstrate different aspects of the difficulties involved and the potential speed gain from it. They are (more details will be given below):

[1] MPI is used to avoid the Python start-up overhead, not for actual parallelism. i.e. There is no communication between processes at all.

[2] trivial MPI refers to the use of MPI mentioned above that has no communication at all.

1. Ground template filter: was originally written in Python with Numpy and C with weave[3]. This is a testbed to see how to fully Cythonize a Python function, i.e. written in C-style with Cython syntax without the use of Numpy at all.
2. Polynomial filter: was originally written in pure Python with Numpy. This is a testbed on the potential speed gain on a function that is highly pythonic and heavily relies on Numpy that full Cythonization is virtually impossible (without reinventing the wheels).
3. Boundary distance function: was originally written in Python with Numpy and C with weave. This is one of the major bottleneck of the whole pipeline, and is the testbed to see how much speed up we could get from Cythonization in an actual application hotspot.

Ground template filter and boundary distance function are both rewritten completely Cythonic without the using of Numpy functions[4]. For the polynomial filter, it is minimally cythonized because of the heavy reliance on Numpy's functions. Since there is virtually no speed gain from the parallelization in the polynomial filter, it will be mentioned in the appendix. In the following 2 sections, we will focus on the ground template filter and boundary distance function. All the codes can be found in GitHub - ickc/TAIL: Experimental pipeline as an alternative to POLARBEAR's AnalysisBackend — Refactorize, modernize, cythonize, parallelize..

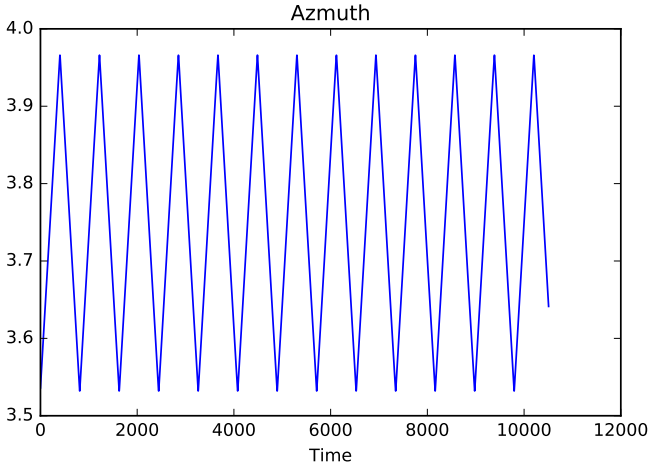
The following benchmarks are done on Cori's Haswell nodes. Although we want to use the KNL nodes to push the vectorization and OpenMP scaling, but Cori's KNL queue has been impenetrable on the days near the submission of this report. In the near future we will profile it on KNL.

3.1 Ground Template Filter

The actual code for this filter can be found in TAIL/ at master · ickc/TAIL/ground_template_filter_array.pyx · GitHub.

3.1.1 Description

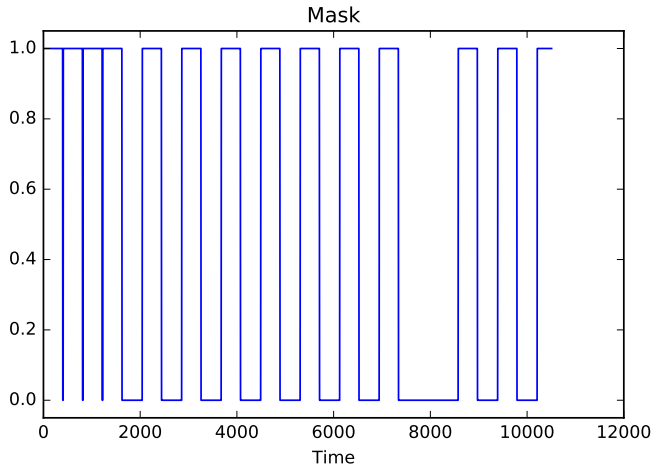
First, we scan the sky in constant elevation with a sweeping azimuth:



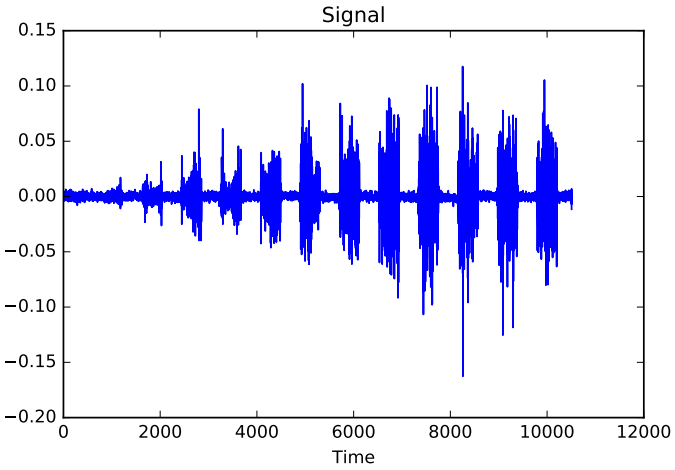
And there is a boolean mask that select the data (and discard bad data):

[3] Weave, formerly Scipy.weave, is a now deprecated tool to allow easy writing of C-modules in Python. Now the Scipy team has migrated their own weave codebase using Cython.

[4] except when creating and returning Numpy array as a Python object



And the signal:



The mask is for data selection. So the main focus here are the signal $d(t)$ and the azimuth $\theta(t)$. These 2 equations can be considered a set of parametric equations in time t. And the ground template filter is basically to find the $d(\bar{\theta})$, the average signal as a function of θ.

In the algorithm, this is done through a given resolution n (say, 300 pixels), the range in θ is divided into n bins, and the signal is averaged into these bins.

3.1.2 Optimization Procedures

3.1.2.1 Vectorization

The first step is to look at the HTML report which can be found in Cython: ground_template_filter_array.pyx, and below we have an excerpt of it:

```
 061:         ## add total signal and no. of hits
+062:         for j in range(nTime):
+063:             if mask[i, j]:
+064:                 k = pointing[j]
+065:                 bins_signal[nBin * i + k] += input_array[i, j]
+066:                 bins_hit[nBin * i + k] += 1
 067:             # the following allows SIMD. But Intel vectorization report sa
 068:             # TODO: try again on KNL
 069:             # k = pointing[j]
 070:             # bins_signal[nBin * i + k] += input_array[i, j] * mask[i, j]
 071:             # bins_hit[nBin * i + k] += mask[i, j]
 072:
 073:         ## average signal
 074:         ## SIMD checked
+075:         for k in range(nPix):
 076:             # won't be 0 since it is initialized as EPSILON
 077:             # if bins_hit[nBin * i + k] != 0:
+078:             bins_signal[nBin * i + k] /= bins_hit[nBin * i + k]
 079:
 080:         # substraction
+081:         if groundmap:
 082:             # no SIMD: report says vectorization here is inefficient
+083:             for j in range(nTime):
+084:                 input_array[i, j] = bins_signal[nBin * i + pointing[j]]
 085:         else:
 086:             # no SIMD: report says vectorization here is inefficient
+087:             for j in range(nTime):
+088:                 input_array[i, j] -= bins_signal[nBin * i + pointing[j]]
+089:     free(bins_signal)
+090:     free(bins_hit)
```

The key to look for in these reports are the "yellow-ness". If there's no yellow-ness seen in the report, it means that it is transpiled into pure C/C++.

The next step is to check for vectorization and see if there is anyway to rewrite it to trigger vectorization. For example, from the same code above, in the HTML version if you click at the + sign in line 65, you can see the C/C++ code it is translated into.

We can then find where this piece of C/C++ code is in the generated `.cpp` file: https://ickc.github.io/TAIL/tail/timestream/ground_template_filter_array.cpp. Trace it back to the nearest for-loop, it is in line 2303.

Finally, you can go into the vectorization report and find the reference of line 2303, and it reads

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization
```

In this case, we will replace the code from line 63 – 66 by 69 – 71. Repeating the whole exercise and the report will say

```
loop was not vectorized: vectorization possible but seems inefficient.
Use vector always directive or -vec-threshold0 to override
```

In this instance, rewriting the code to remove the branching statement made vectorization possible, but still inefficient and the compiler didn't vectorize it. It is worth nothing that the code in line 69 – 71 is slower than the one in 63 – 66. Without checking the report, one would write using line 69 – 71 to hope for vectorization but actually get a slower code.

In other instances of the code, such kind of optimization paid off. One such case is to use the Conditional (ternary) Operator to rewrite an if-statement.

Another interesting instance in this particular problem is

```
for i in range(size):
    bins_hit[i] = EPSILON
...
## SIMD checked
for k in range(nPix):
    # won't be 0 since it is initialized as EPSILON
    # if bins_hit[nBin * i + k] != 0:
    bins_signal[nBin * i + k] /= bins_hit[nBin * i + k]
```

In order to remove the special case of dividing by zero, `bins_hit` is declared to be `double` instead of `int`, and initialized to be the machine epsilon rather than 0. In the worst case scenario it makes an error of the order 10^{-16},

but is going to speed up this part of the code a lot. Since we know that in our application we never will achieve this level of precision, this will be a good trade off for us.

Of all the vectorization effort, one feature we missed is the ability to control the alignment of the memory. Numpy array are created in 16-bytes alignment, which will be provided as the input of the function. But in order to fully optimized for vectorization, 64-bytes alignment is needed. Even worst, even if it is 64-bytes aligned[5]), directive pragma is not support in Cython and therefore there is no speed up from alignment.

3.1.2.2 Locality

In this example, it is also worth mentioning locality is also important. It is because the computation is of low intensity, the problem is IO bound and we want to minimize the time spent in IO.

From the description above, we see that the $\theta(t)$ is sweeping back and forth, and the signal $d(t)$ is averaged per bin in θ. i.e. as it is swept through, either you have the locality of $d(t)$, or the locality of the bins of θ, but not both.

Furthermore, this function has a key `lr`, when true, the averaged signaled is calculated separately when θ is sweeping left or right. In the original version of the code, this is done in 2 passes, where each pass has a mask to remove the signal that move in another direction (e.g. for left pass, right moving θ are removed by a mask). This is going to be RAM inefficient and slow because the IO on the signal is twice as long and require a mask as large as the original signal.

When such conflicts exists, the locality is prioritized towards the signal rather than the bins. Because the total number of bins is much less than that of the signals, so the bins has a higher chance to stay in a higher level of cache (say L3), then the signals which is typically larger than L3 cache. One such example is this code:

```
for i in prange(nCh, nogil=True, schedule='guided', num_threads=num_threads):
    # calculate ground template
    ## add total signal and no. of hits
    for j in range(nTime):
        if mask[i, j]:
            k = pointing[j]
            if isMovingRight[j]:
                bins_signal_r[nBin * i + k] += input_array[i, j]
                bins_hit_r[nBin * i + k] += 1
            else:
                bins_signal_l[nBin * i + k] += input_array[i, j]
                bins_hit_l[nBin * i + k] += 1
```

In the if-statement above, priority of locality is given to the `input_array`. Otherwise the 2 branches should be split into their own for-loop.

3.1.2.3 OpenMP

The same example above also shows the use of OpenMP. The `nogil` is important here. Only region with no "yellowness" in the Cython to C/C++ conversion can releases the Python Global Interpreter Lock (GIL). So if one wants to use OpenMP parallelization, then the content within the loop needs to be free of Python interaction.

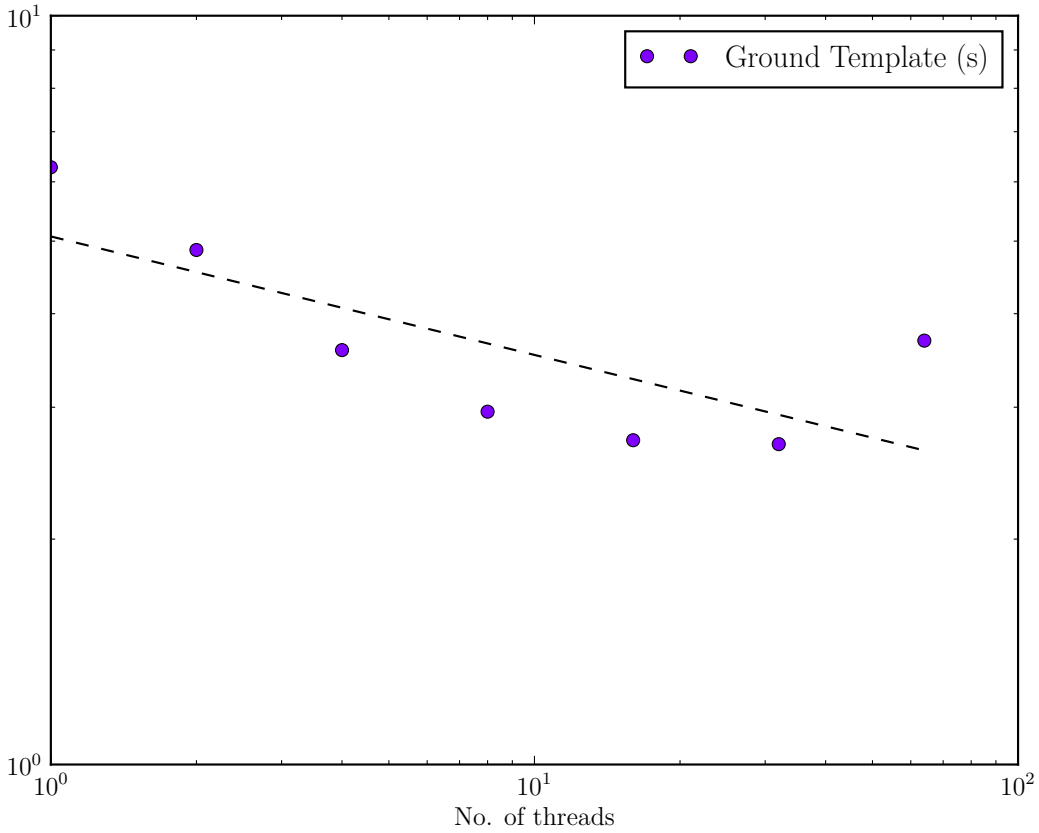
This example also shows that the `i`-loop that runs through the `nCh`, stands for the number of channels, are independent. This is often true in the filters and will be trivial to parallelized.

3.1.3 Benchmark

With number of channels 100, number of time-stream signal 10000, and number of bins 300, `lr=True`, the ground template filter in the old pipeline takes 7.53 ms to complete, and our code takes 1.96 ms to complete. i.e. a speed up of 3.842 times, just from Cythonization and SIMD *without* OpenMP parallelization yet.

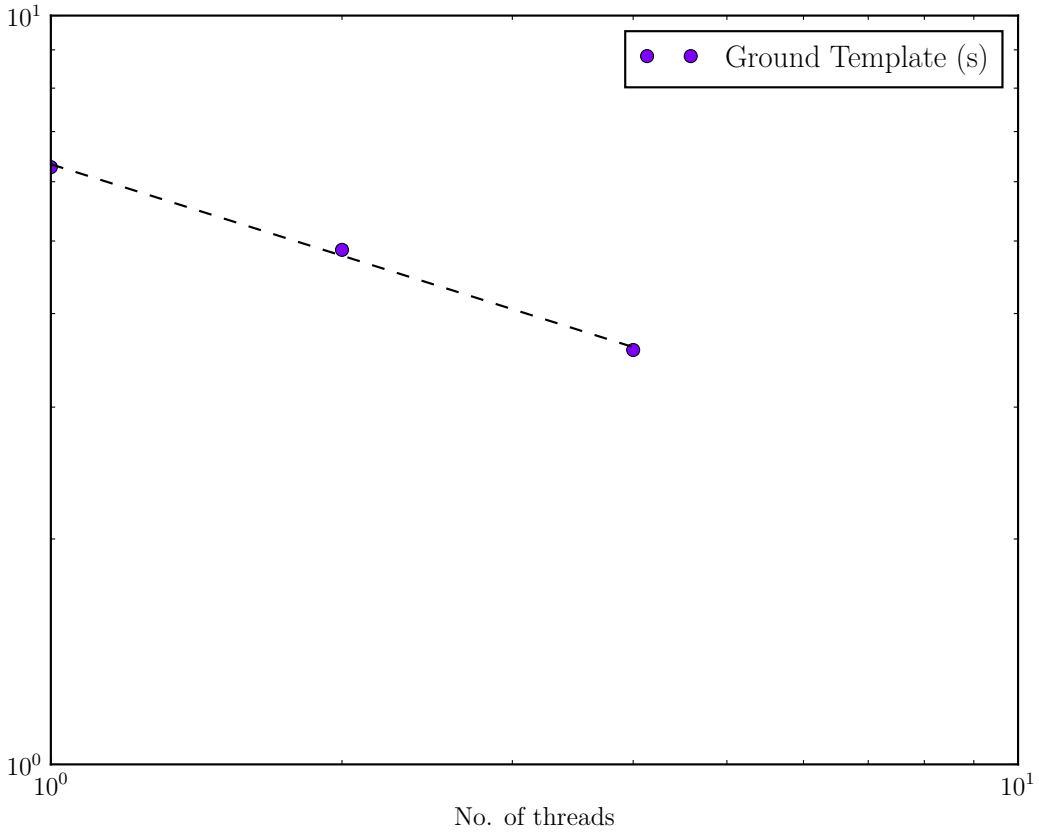
[5] which can be achieved by copying the data, `malloc`, and some pointer arithmetic, all available in Cython (but `aligned_alloc` is not).

As we add more numbers of concurrent threads, the scaling looks like the following, using number of channels 50,000, number of time-stream signal 10,000, and number of bins 8,192, `lr=True`:



It can been seen that this does not scale very well beyond 8 threads, and hyper-threading (not plotted here) actually will make it slower. The reason is that even for this unrealistically large no. of channels, the computation intensity is still very low and the OpenMP overhead is relatively high when p is large — it is likely that all threads are accessing the L3 cache where the bins mostly lived and the RAM where the signal mostly lived and are congressed. i.e. in the roofline model we are on the way left side of the roof.

Fortunately, since we are starting 16 processes. When Haswell/KNL nodes are used, the number of threads is 2/4 respectively. In this region,



we see that the strong scaling coefficient is 0.406, meaning that the time needed scales as $\frac{1}{p^{0.406}}$. As explained above, the low coefficient is explained by the low computational intensity.

3.2 Boundary Distance Function

The actual code for this filter can be found in TAIL/boundary_distance.pyx at master · ickc/TAIL.

3.2.1 Description

This function is much easier to be described, and involve well-known algorithm in Computer Science: boundary tracing algorithm and post-office problem.

The function takes a $n \times n$ 2D array, which has 1 simply-connected region of `True` value, and elsewhere `False`. It can be visualized as 1 single island of `True` value around a sea of `False` in a square map[6].

What this function does is for every point in the island, find the distance to the closest point on the boundary. A 2D array of these distances are returned as a Numpy array.

We implemented it by first using a Boundary tracing algorithm to detect the boundary first. After that, it is a post-office problem — for every point inside the boundary, determine the closest point on the boundary.

[6] Physically, the `True` island represent the map we have for the CMB. Since we are scanning for a curvilinear sky, the map we made is not necessarily rectangular.

3.2.2 Optimization Procedures

The vectorization and OpenMP parallelization procedure is the same as that laid out in Ground Template Filter and we are not repeating it here. The Cythonization report is in Cython: boundary_distance.pyx (and you can see most Python interaction is removed), and the generated `.cpp` code is in https://ickc.github.io/TAIL/tail/timestream/boundary_distance.cpp, and the vectorization report is in here.

One example that shows both of these is

```
for i in prange(x_min + 1, x_max, nogil=True, schedule='guided', num_threads=num_threads):
    for j in range(y_min + 1, y_max):
        if mask[i, j]:
            loc = i * m + j
            # SIMD checked
            for k in range(nBoundary):
                distance_sq = (i - boundary_coordinate[2 * k])**2 + \
                    (j - boundary_coordinate[2 * k + 1])**2
                distances_sq[loc] = distance_sq \
                    if distance_sq < distances_sq[loc] else distances_sq[loc]
```

This is the part the function is going to spend the majority of time in when determining the distances of each points to the boundary because it is $O(n^3)$. In the innermost `k`-loop, it is vectorized using the ternary conditional operator, without which the if-statement would have prevented vectorization. In the outermost `i`-loop, OpenMP is used, where `i` represent the `i`-th row of the pixels on the map. This can be easily parallelized because the `i`-th row are independent of each other.

Another (counter) example is

```
# convert boundary from 1D indexing to 2D indexing
# and obtain the smallest box that includes the boundary
# Not vectorized, use the following if in C
#pragma ivdep
for k in range(nBoundary):
    x = boundary[k] // m
    y = boundary[k] % m
    boundary_coordinate[2 * k] = x
    boundary_coordinate[2 * k + 1] = y
    x_min = x if x < x_min else x_min
    y_min = y if y < y_min else y_min
    x_max = x if x > x_max else x_max
    y_max = y if y > y_max else y_max
```

Because of the apparent vector-dependance, it is not vectorized. If it were written in C/C++, we would have used a `#pragma ivdep` at this point. Fortunately it is $O(n)$ and is not the hotspot of the application.

It is worth mentioning that the boundary tracing algorithm is not parallelized nor anywhere has SIMD vectorization in it. This part of the code is essentially having a class of object `Turtle` that walks down the map and keep tracing the boundary until it goes back to the starting point.

Vectorization is impossible in this algorithm, but OpenMP parallelization is at least in theory possible. We could have divided the map into p regions and starts "workers" to trace the boundaries in parallel and later merge them together. However, it will becomes non-trivial to merge the individual boundaries together to make it becomes one big boundary, while remain robust.

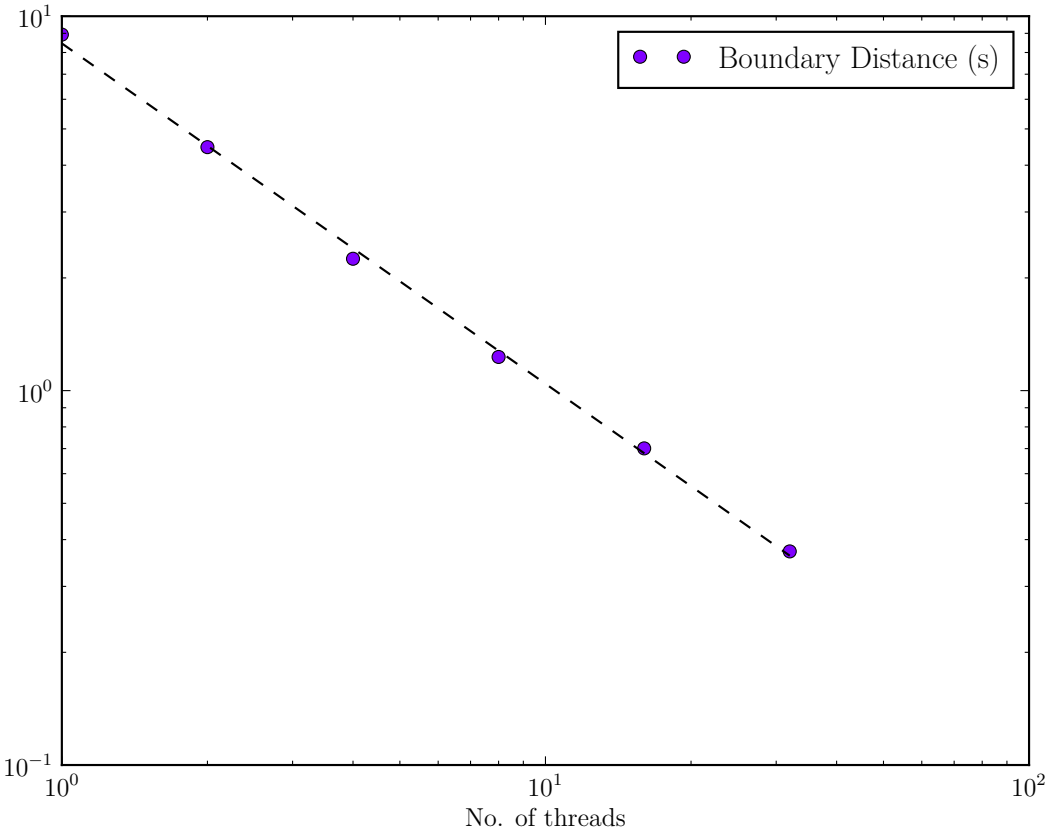
The reason we choose not to parallelize this region is because this algorithm is $O(n)$ (bare in mind that the input mask is of size $n \times n$), and since it is not optimizable, potential speed-up from even KNL is at most 68 times (and in reality very much less because we are starting 16 processes at once), without even account for the time to combine all those boundaries.

However, we need to stress that this possibility is not ruled out. If in the future this becomes a hotspot, such parallelization scheme should be investigated. In fact, although we wrote this part of the code using object-oriented programming, we used Cython's Extension types (aka. cdef classes) such that they are not Python classes but are translated into structs and functions. i.e. All Python interaction is removed and GIL can be released, which is the cornerstone of using OpenMP in Cython. A bonus of this design is that there is no Python overhead and the algorithm is very fast.

3.2.3 Benchmark

We created a mask of 1024×1024 large, and an island of radius 128 in the middle. We run the benchmark on the original AnalysisBackend algorithm at 583 ms, and our algorithm at 119 ms, i.e. a speed up of 4.90 times, just from Cythonization and SIMD *without* OpenMP parallelization yet.

In the strong scaling test, we changed the radius to 510, and the result is



we see that it is almost perfectly parallelized with a strong scaling coefficient of 0.909, meaning that the time needed scales as $\frac{1}{p^{0.909}}$. This is close to ideal (1), and is expected because the channels are totally independent and hence is perfectly parallel.

4 Conclusion

Speedup is achieved through

1. Cythonization
2. SIMD vectorization

3. OpenMP parallelization

It is hard to decouple the 1st and 2nd effect. For the combined effect of Cythonization and SIMD vectorization, in the worst case scenario which involved minimal changes to the Python code hence remains Pythonic, the Polynomial Filter Array in the appendix, has 3.23 speedup. In the best case scenario where Python interaction is minimized without using Numpy functions, the Boundary Distance Function, a speedup of 4.90 is achived.

For the 3rd effect, Polynomial Filter Array which stays Pythonic and relies on Numpy has virtually 0 efficiency. More investigation is needed here, more is mentioned in the appendix. For both the Ground Template Filter, the strong scaling coefficient is 0.406 because of low computational intensity, and fortunately is not the hotspot of the application[7] Lastly, in the Boundary Distance Function, high strong scaling coefficient is achieved at 0.909. This will be important since this is a hotspot for the application.

To give a very rough estimation on the potential speedup for the whole pipeline, we will give a factor of 4 for Cythonization and SIMD vectorization. We guesstimate the strong scaling to be ~0.5, with a huge grain of salt that it varies a lot in these 3 different functions. From this preliminary study, hyper-threading does not help and so we are not considering this. Since we use 16 process per node, for Cori's Haswell nodes, we would use 2 threads per process, and for KNL, we would use 4 threads per process. The overall estimated factor of speed up will then be:

1. Haswell nodes: ~ 6
2. KNL nodes: ~ 8

For a job that would have required $\sim 220,000$ NERSC hours on Haswell, it would becomes $\sim 40,000$. It should be emphasized that this is a very rough estimation though.

A More on Boundary Distance Function

The boundary distance function is actually part of a larger code called the pseudo-power spectrum. Boundary distance function is needed for an apodization mask, with essentially filter the map such that the "island" has a smooth transition to zero on the boundary.

This apodization mask might be created by an $O(n)$ algorithm instead (whereas the boundary distance function is $O(n^3)$), essentially by walking through the boundary k-times, where k is the width of the apodization mask.

This has not been done in this study, because we find it very hard to parallelized a boundary tracing algorithm.

However, this alternative proposal on the apodization mask might turns out to be more effective because as the resolution of the map we make gets higher and higher, $O(n)$ is always going to win the $O(n^3)$ algorithm.

More investigation is needed in this comparison. And in fact a quick demo based on the implementation of boundary distance function in this report is possible. Because a boundary tracing algorithm in this report is already implemented, and the proposed apodization mask is basically an iteration of the boundary tracing k times.

B Polynomial Filter Array

B.1 Description

The poly_filter_array code uses a Legendre polynomial to fix the gain of the data, as the gain drifts during the scanning period. Unlike ground_template_filter_array , poly_filter_array relies heavily on Numpy functions, which for pure Python code provides good speedup. However, it is very unfortunate when one is trying to transition the code to Cython.

To transition the code to Cython, we rewrote the code in the Cython memoryview array style, similar to the ground template filter. However, when it came time to parallelize, we ran into serious problems. To parallelize Cython code, it is required that we release the GIL, which is not possible when code contains Numpy calls. Unfortunately, this was almost the entire code, which meant that the few cases where we actually could call prange resulted in no speedup whatsoever. For a future direction, we recommend rewriting the code to eliminate the use of Numpy, which would enable the code to be reformatted in C-style and parallelized, which is the maximum Cython speedup.

[7] in actual data size, it finishes on the order of ms.

One more point of interest with this code is that it was written and compiled in a Jupyter notebook that was run within NERSC. NERSC uses the Intel MKL distribution of Python and Numpy, which actually already uses OpenMP to deliver thread-level parallelism and include specialized vectorization instructions. This means that simply by compiling and running the code on NERSC, we may already be taking advantage of parallelism that's implicitly built into the Python framework.

In summary, when code contains significant amounts of Numpy operations, optimal conversion to Cython and Cythonized parallelization is not possible. To fully implement code in Cython and to parallelize it, code must be rewritten to eliminate Numpy calls.

B.2 Benchmark

For number of channels 10,000, number of time stream data point 10,000, and polynomial order 4:

In the original code, it took 1.14 s and in the cythonized code, 353 ms in serial. i.e. 3.23 times improvements.

However, there is virtually no difference in timing when number of threads is increased. This is unexpected because even if the function is heavily depends on the Numpy functions, Intel's distribution of Python supposed to have built Numpy with OpenMP and SIMD optimization.

More investigation is needed here in the future, and perhaps requires a deeper understanding of what kind of OpenMP parallelism is provided in Intel's Distribution of Python.

Because there is no parallelization scaling in this function (the coefficient is virtually 0), we decided not to put this part in the main report but in the appendix.

C Numba

One of the options that we pursued for code speedup/parallelization was Numba. Numba is a module that generates optimized machine code from a python codebase using the LLVM compiler structure. LLVM is a single static assignment(SSA)-based compilation structure. It allows you to just-in-time compile Python code (including Numpy) at import time, runtime, or statically. To test the speedup, we ran a nontrivial test function; the Numba additions are in red (jit is just-in-time compiling).

```
import numpy as np
from numba import jit

@jit
def naive_convolve(f, g):
    vmax = f.shape[0]
    wmax = f.shape[1]
    smax = g.shape[0]
    tmax = g.shape[1]
    smid = smax // 2
    tmid = tmax // 2
    xmax = vmax + 2*smid
    ymax = wmax + 2*tmid
    h = np.zeros([xmax, ymax], dtype=f.dtype)
    for x in range(xmax):
        for y in range(ymax):
            s_from = max(smid - x, -smid)
            s_to = min((xmax - x) - smid, smid + 1)
            t_from = max(tmid - y, -tmid)
            t_to = min((ymax - y) - tmid, tmid + 1)
            value = 0
            for s in range(s_from, s_to):
                for t in range(t_from, t_to):
```

```
                v = x - smid + s
                w = y - tmid + t
                value += g[smid - s, tmid - t] * f[v, w]
        h[x, y] = value
return h
```

In normal python without Numba, this code takes 1.41s to run with a 100x100 and 8x8 array for f and g. With Numba and jit, this code takes 2.8 ms to execute, a speedup of more than 40x, but times depend on the machine used. It has been reported that Numba can speed code up up to 200x (https://en.wikipedia.org/wiki/Numba).

When we began looking into Numba, we were under the impression that there was an additional function called prange, which uses as many CPUs as detected by the multiprocessing module and runs loops in parallel. However, we then found out that this function had been removed and Numba no longer possesses parallelization options. At this point, we abandoned this strategy, but it is worth mentioning because of the incredible (non-parallel) speedup obtained for Python code with very minimal effort.

D Intel TBB, MPI

In the situations that involve more complicated parallelization, MPI and Intel Threading Building Blocks (TBB) might be beneficial. We did not try either approach here, but Intel has some nice articles on using MPI and TBB in Python:

- Unleash the Parallel Performance of Python* Programs | Intel® Software
- Exploring MPI for Python* on Intel® Xeon Phi™ Processor | Intel® Software

In particular, Intel TBB can becomes important if different levels of parallelization is applied, and oversubscription might arise. Currently, all 3 functions involved in this study has incorporate a key-value function argument of `numthreads` default to be 4. If function composition is used, these `numthreads` should be set properly to avoid oversubscription. But the alternative will be setting `numthreads` with the environment variables `OMP_NUM_THREADS` and uses Intel TBB to get around with oversubscription whenever it arises.

Reference

Collaboration TP, Ade PAR, Akiba Y et al (2014) A Measurement of the Cosmic Microwave Background B-Mode Polarization Power Spectrum at Sub-Degree Scales with POLARBEAR. arXivorg 171.