# Hardware and Software Used

Hardware: G2.2xlarge Amazon Instance using NVIDIA GRID K520 GPUs.

<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/accelerated-computing-instances.html>

Software: Numba 0.35 with Python 3.5.2

# Testing and Helper Code

To run code:

runCode(1,100000);

## runCode(numba\_flag, scaling)

numba\_flag = 1/0. If 1, it will try to run the Numba versions of each function

scaling = multiple of the input file row count to use

## importData()

The main thing to note here is the transpose. As with SAS code, the transpose brings all grant info onto one record. This is necessary to ensure that the main Numba loops take advantage of the “embarrassingly parallel” nature of the problem. The main loop simply iterates through each record with no computations dependent on the results of another iteration.

The number of days is also calculated here. This could potentially be pushed into Numba. If the input data is coming from SAS, handiest to use the raw SAS date (days since 1960) as then no date transformations required within Python and Numba can work away on the numeric field.

Finally, it may be necessary to regularize the time intervals between vestment dates to the nearest number of months. This could be done in the input file or during this processing. Reason for this is that small variations could change the slopes enough to allocate the line segments differently.

# Algorithm 1-Calculate Cumulative Value for each Tranche

## Jit Function

In: row counter (for input dataset), number of Tranches per grant, the shares and values Numpy arrays for input and the cumulative vestment Numpy array for output

Out: Numpy Array “cumVest” which is then used to update the cumulative vestment columns of the data frame

@numba.jit(nopython=True)

def cumVestWrapperN(rowCount, numTranches, shares, vals, cumVest):

#This is a hot loop. No dependence between obs so Numba should work in parallel

for i in range(0,rowCount):

total=0

# GET CUMULATIVE VESTMENT AMOUNT PER GRANT FOR EACH TRANCHE

for j in range(0,numTranches[i]+1):

total+=shares[i,j]\*vals[i,j]

cumVest[i,j]=total

return cumVest

### Results

500x speed-up with Numbas

Function: cumVest duration (seconds):1.5758375410000554

With Numbas: Function: cumVestN duration (seconds):0.0037875410000651755

Note: The @numba.jit(nopython=True) option is useful as it forces a failure if any of the Numba code cannot be converted to Numba. Otherwise, will revert to “object code” which is as slow or slower than not using Numba.

GuVectorize Function

@guvectorize(['void( int64[:], int64[:], intp[:],intp[:], int64[:])'], '(n),(n),(),()->(n)', target='cuda')

def cumVestWrapperGU(shares, vals,rowCount, numTranches, cumVest):

total=0

# GET CUMULATIVE VESTMENT AMOUNT PER GRANT FOR EACH TRANCHE

for j in range(0,numTranches[0]+1):

total+=shares[j]\*vals[j]

cumVest[j]=total

You can use Vectorize to work over vectors and GuVectorize to work over matrices. The looping over rows is therefore implicit and the dependencies between columns can be dealt with through looping within the function.

Function: cumVestGU duration (seconds):0.039265901999897324

The GuVectorize version of this function is 10x slower than @jit.

# Algorithm 2-Allocate Segment Number to Tranche

## Numba Functions

3 functions are used for this algorithm. The first step is the hot loop and loops through at a record level, guaranteeing parallelization. The next two functions perform the operations per row. See comments inline for more detail.

@numba.jit(nopython=True)

def segWrapperN(rowCount,n, numTranches, segments, days, cumVest):

#This is a hot loop. No dependence between obs so Numba should do in parallel

for i in range(0,rowCount):

#Create an empty vector to hold the result for this grant

tmpSeg = np.array([ [ 0 for x in range(n) ] for y in range(1) ])

#End is number of tranches for this grant (differs per grant)

end=numTranches[i]+1

#Update the segment numbers for this grant

segments[i,:] = getSegmentsOuterN(end,cumVest,days, i, tmpSeg)

return segments

@numba.jit(nopython=True)

def getSegmentsOuterN(end, cumVest,days, i, tmpSeg):

start = 0

seg=0

#Will run getSegmentsInner from first to last tranche on the first go

#Then keep running it through until all tranches assigned segments

while start < end:

#The inner loop establishes a segment and returns the last tranche for the segment

last = getSegmentsInnerN(start, end, cumVest,days, i)

#As this is a new segment, the segment number increases

seg=seg+1

#Allocate the segment number to all tranches in the segment

tmpSeg[0,start:last+1]=seg

#Next iteration will start from the next unallocated tranche

start=last+1

return tmpSeg

@numba.jit(nopython=True)

def getSegmentsInnerN (start, end, cumVest,days,i):

maxSlope = 0

#The first tranche is a special case as slope is just y/x

for j in range(start,end):

if j==0:

tmpSlope=cumVest[i,0]/days[i,0]

#Otherwise calculate the slope as (y2-y1)/(x2-x1)

else:

tmpSlope=(cumVest[i,j]-cumVest[i, j-1])/(days[i,j]-days[i,j-1])

#Keep a record of the max slope so far

if tmpSlope > maxSlope:

last=j

maxSlope=tmpSlope

#Return the last tranche of the max slope line - this is the end of the new segment

return last

### Results

180x speed-up with Numba

Function: segWrapper duration (seconds):18.483691285000077

Function: segWrapperN duration (seconds):0.11254592499994942

# Discussion of Results

I created a gpu implementation of cumVestWrapper using guVectorize. The numba jit outperforms the cuda implementation by a factor of 10. Research indicates that the algorithms here may not be best suited to a GPU implementation. There’s no linear algebra or numeric operations going on that would max out the processors.

<https://stackoverflow.com/questions/41804003/numba-guvectorize-barely-faster-than-jit/41810390>

And <https://stackoverflow.com/questions/35350689/numba-guvectorize-target-parallel-slower-than-target-cpu>

“Generally speaking, you will see much more benefit from the parallel ufunc target if you are doing more math operations per memory element (like, say, a cosine).”

Finally, GPUs are not great at processing a lot of looping and branching. The segment assignment algorithm consists mainly of loops across the elements. It may be even less suitable for GPU implementation.

There is an option to use a lower level entry point to CUDA such as @cuda.jit. It may be possible, through manual control of memory and other resources, to deliver more efficiencies. However, if the issue is, as above, due to a mismatch of problem to architecture, then it won’t help. This optin would also mean moving further away from plain Python code and into maintaining a lot of plumbing code around the business logic.

For those reasons I would recommend one or more of the following options.

1. Use @numba.jit. It provides significant speed-up which may be enough for your particular needs and doesn’t require creating a lot of extra plumbing code.
2. Create a Pandas/Numpy implementation of the algorithm using whatever high-level features might be of use. A lot of the Pandas and Numpy features are already optimized under the hood so, for use cases that can be implemented by these, there is no great advantage of moving to plain python in a numba function to get the C/Fortran style speed-up. The amortization algorithms require iteratively working across rows which doesn’t map so well to the standard pandas/numpy functionality so the plain python implementation in numba may well offer a speed-up, in this case. The point is though, that comparing plain python to plain python with numba is not the best comparison. Best to avail of the python library features of pandas/numpy/other to get the algorithm working and then compare that to the plain python implementation using numba to see if numba is giving a boost. If they are close, then better to use the former as the code using the high level constructs will be easier to maintain and improve over time.
3. Most high-performance data processing pipelines I’m familiar with are now implemented on MapReduce or Spark. The architecture and programming framework is designed for highly parallelizable data processing. If you have access to a cluster, it may offer a viable alternative to GPUs. As this is fast becoming the standard, there are no shortage of tools and developers around.
4. If you want to go deeper with the GPU experimentation, you may want to engage a specialist. As Python developers with GPU experience are hard to come by, a good video game developer may be a good bet. Even if they are not already familiar with Numba, they will grasp that part quickly and will be able to advise further if your business case can be mapped well to the GPU architecture.

# Questions:

         I am still getting used to 0-based arrays, so I will spend a little bit of time unrolling/printing all of the values during a small execution set

I recommend using the Anaconda python distribution and its IDE Spyder. It has great debugging including a Variable Explorer. You can step through the code line by line and look at the contents of each array and variable as it changes.

         I see the numba.jit is not using parallel=True (or that we are not using cuda.jit or vectorize)

o   I thought that numba.jit is just optimized machine code (to try to get C++/Fortran speed), not parallel code.  What am I missing?

* I am most interested in getting this to run across a CUDA GPU.  Is it possible to update this to run on a GPU

Please see discussion section above for this.

         When I see “col.startswith” it makes me nervous.  Is there a better way or do I just need to make sure my vars do not start with the same thing?

The idea here is, like with SAS, to group the variables in some dynamic way rather than be tied to keeping the same column order all the time in the input file. So if you use a standard naming convention for your variables this will work nicely. There are alternatives such as using the column number or reading in a metadata file that tells the program the column groups. But I find using prefixes to be the most maintainable solution.

         I am not getting the same performance gains as you, but I am using a ~5 year old AMD processor (15x on cumVest and 24x on segWrapper).  Does that pass your sniff test?

I think that speedup is very much dependent on the hardware setup.

         Are the shares, vals, and days numpy series actually copying the data from the dataframe or only referencing it?

They are creating copies.

         Can you show me what the days calc would look like using Numba?

If the dates come in as just raw SAS dates (days since 1960) it would just be a case of a-b in @jit function. So it would be something like below. But, as per discussion above, might not be the most efficient thing to do.

@numba.jit(nopython=True)

Def calcDays (vestDate, grantDate)

Days=grantDate-vestDate

Return days

         The pivot table trick is pretty slick.

Yes, it is! It’s funny because is such a well-known procedure in the SAS world and I’ve seen a bunch of questions on StackOverflow asking how to do it in Python met with complete miscomprehension.

         There are some edge cases surrounding date diffs that I probably should have put in the test data.

o   When subtracting grant date from vest date, we always add one (because the vest date could equal the grant date, and in that case we expense over 1 day)

* Two tranches could technically have the same vest date (and we need to have a special case in the getSegmentInner logic to avoid a divide by zero issue)

If it’s a case that the gap between tranche can be considered as some standard measure then that would be more robust. i.e. 1 month +- a day or two for example. If they have the same day, I guess could also merge the tranches into one for purposes of the program.

* I generally put in a ton of error/exception throwing to make sure issues do not propagate through a calculation further than they should (negative slopes, negative days, etc).  What is the best practice for doing this in Python?

The best way to handle this would be with assertions as below. The handy thing is you can turn them off if you want to do a faster run as per <https://stackoverflow.com/questions/1693088/what-is-the-use-of-pythons-basic-optimizations-mode-python-o>

assert 2 + 2 == 5, "Houston we've got a problem"

### 