# 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

## Numba 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.

# 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

### 