5-2015

Efficient Parallel Approaches to Financial Derivatives and Rapid Stochastic Convergence

Mario Harper
Utah State University

EFFICIENT PARALLEL APPROACHES TO FINANCIAL DERIVATIVES AND RAPID STOCHASTIC CONVERGENCE

by

Mario Y. Harper

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Financial Economics

Approved:

Dr. Tyler Brough Major Professor	Dr. Jason Smith Committee Member
Dr. Jared DeLisle Committee Member	Dr. Richard Inouye Associate Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2015

Abstract

Efficient Parallel Approaches to Financial Derivatives and Rapid Stochastic Convergence

by

Mario Y. Harper, Master of Science

Utah State University, 2015

Major Professor: Dr. Tyler Brough
Department: Economics and Finance

This Thesis explores the use of different programming paradigms, platforms and languages to maximize the speed of convergence in Financial Derivatives Models. The study focuses on the strengths and drawbacks of various libraries and their associated languages as well as the difficulty of porting code into massively parallel processes. Key languages utilized in this project are C++, Python, Java, C, and Matlab. Along with these languages, several multiprocessing libraries such as C++ AMP, Python NumbaPro, Numpy, Anaconda, Finance Toolkit and such were compared. The results of the development and implementation of code suggests that once we massively parallelize the operations, the only difference in speeds accross systems are found only in the unique language's overhead. The findings also suggest that in terms of development and testing time, one should choose to utilize a language they are most familiar with rather than to try a minimalistic approach using a more base level language.

(52 pages)

Public Abstract

Efficient Parallel Approaches to Financial Derivatives and Rapid Stochastic Convergence

by

Mario Y. Harper, Master of Science

Utah State University, 2015

Major Professor: Dr. Tyler Brough
Department: Economics and Finance

Finance is largely about how quickly one can act. When an opportunity comes, the best time to act is immediately. However, with complex financial instruments, it is difficult to quickly come to a solid conclusion. This study is about how different computational methods were used to speed up the process of pricing these difficult instruments. On top of that, some neat math is used to make the priced results more accurate.

To my wife Michaela who was willing to put up with me, my computers, and our baby girl who tried to take the aforementioned computers apart.

Contents

List of Tables

List of Figures

Acronyms

APU	Accelerated Processing Unit
CPU	Central Processing Unit
GPU	Graphical Processing Unit (Video Card)
CUDA	Compute Unified Device Architecture (parallel computing platform)
DE	Development Environment
ENIAC	Electronic Numerical Integrator and Computer
I/O	Inputs and Outputs
SDE	Software Development Environment
SDK	Software Development Kit
JVM	Java Virtual Machine
JRE	Java Runtime Environment
AMP	Acellerated Massive Parallelism

Chapter 1

Introduction

1.1 Significance

One who wishes to take advantage of an arbitrage position must be quick, this is true in any branch of finance. To find an under or overpriced option requires rapid computation of sometimes complex stochastic algorithms in quick succession. In essence, one must first find information, process it, calculate a position to take, and excecute that position in rather quick succession. Obviously not an easy task.

As has been widely noted [3], both traders who appear informed and uninformed about future prices experience a cost induced from latency. Speed is an integral part in pricing and trading for any financial market [4].

The premise of this work is that utilization of compuational tools coupled with the correct programming can vastly cut down convergence times as well as increase accuracy of the pricing. These methods are also designed to reduce the time to implementation for the programmers as well. Depending on the elegance of the implementation, we can shrink the convergence times to 1/3000th of the original convergence time.

1.2 Background

In ye olde days, (that is, right around 2004) serial processes made the bulk of algorithms. Serial processes are tasks that are done on one processor sitting on one CPU. These algorithms run on one resource and have been refined from the begining of computing and have become very quick in excecution time. CPU's have also become much quicker over time allowing these serial processes to attain speeds that I am sure the ENIAC would have never dreamed of. However, as time has gone on. The speeds that an individual processor

could achieve has not increased as quickly as in times past irrespective of Moores' Law. This is easily seen from the quantum thermodynamics involved in the transistor.

1.3 Transistor Dynamics and Quantum Effects

The transistor is an amazing piece of quantum engineering. This is ubiquitous to all electronics because it is the fundamental building block of electronic logic. Transistors can be used as a switch to either be on or off. By placing them into a simple logic circuit, one can create gates such that current can only flow if transistor A and B are on called 'And' gates. Many gates exist, primarily 'And', 'Or' and 'Nor' gates. This can be put together to form small current holdings called a 'bit' which can be put together to form the memory storage that everyone knows. To see the issues related to a CPU, we must first understand the transistor and its limitations. This can be seen in the following example.

We can use the specs of the machine that was primarily used for the work on this thesis. This machine has an intel core i7-3770K processor, this processor was created using atom-by-atom layering construction such that the fabrication size per transistor is 22nm. In laymans terms, there are only about 66 atoms (Silicon atoms are .31 and the doping agent atoms are anywhere from .27 to .43 nm in length) per transister in this processor. Multiply this by 1.4 billion (the total number of transistors on this chip) and we have the approximate number of silicon atoms. This is all packed into a box that is $113mm^2$ by $45mm$. It is readily observed that when these transistors are placed under a high stress computation, there is going to be some thermal energy generated (not to mention quantum effects within the transistors).

In running some basic stress tests, the CPU temperatures reach 54 degrees in Celsius. Coupled with a comfortable 21 degrees for room temperature, the CPU ran at 75 degrees. Temperature being simply viewed as the aggregate kinetic energy from electron collisions, we can see that at this temperature, silicon atoms are likely to be somewhere is the third or fourth energy states. This of course raises real issues, silicon atoms are called semi-conductors because they do not conduct electricity until a threshold energy at which time they become conductors. Although electrical energy is more efficient at electron excitation,

thermal energy does the same thing. There exists a critical temperature threshold at which the electrons will spontaeneously conduct and the processor experiences catastrophic failure. Heat will also deteriorate the oxide layer between transistors causing leaks into swaths of transistor cells, heat is a natural result of electron movement but has become a more critical issue as the size of transistor decreases.

The implication is obvious that as more computation is done, or put in physics terms, as electron states of the atomic clusters are rapidly fluctuated, higher ordered physical constraints have significant performance impact. The more work the transistors do, the hotter they get. The hotter they get, the less efficient they are as electrical signal gets lost in thermal noise (Remember that both electricity and heat are the same electrons). In fact we can express this as a transcritical bifurcation of the cannonical form: $\frac{dx}{dt} = rx(1-x)-\alpha x$ where we can easily see that the new equilibrium must fall into a free conductor.

The fundamental lesson that we learn from the transistor is that there exists a limit to how fast one can be without sacrificing accuracy and the safety of the component.

1.4 Multiprocessing

To increase the speed of tasks and prevent the heat-death of any processor while not sacrificing the performance, computers from 2004 have nearly all had more than one core or processor. A processor handles a process, obviously. A processor usually shares sytem I/O and the systems memory. A system with multiple processors can utilize them to his advantage by handing tasks off to each one to do independently and consolidating the results in the end. Thus, all one has to do is to split tasks into independent processes to realize the benefit of this kind of system.

Imagine if you will, a shipping firm that must deliver parcels accross a reasonably sized population. What might take one truck an entire days work may take two trucks a little more than a half days work. However, the more trucks that one accumulates, the more infrastructure is needed to direct and apportion the truck. There is also a need to guage how much load each truck can take, where the boundaries of each truck should be, effective ranges of the trucks and their difficulties of point access. Thus lies the difficulty of parallel

programming with multiple processes.

The difficulty is incured in largely two places. Firstly, parallel code has a new set of unique problems that can occur (as compared with serial code) and second, speed increases will tend to follow Amdahls Law. Amdahls law simply states that the more you can put your code in parallel, the greater the speedup that comes from the number of processors that you can utilize. In an ideal world, two processors is twice as fast as 1. Unfortunately, this relationship is (as everything appears to be) increasing at a decreasing rate. We generally see that the relation tapers off once you reach about a thousand processors even for highly parallel code. Speedup is not perfectly linear.

Amdahls Law is illustrated in Fig. 1.1

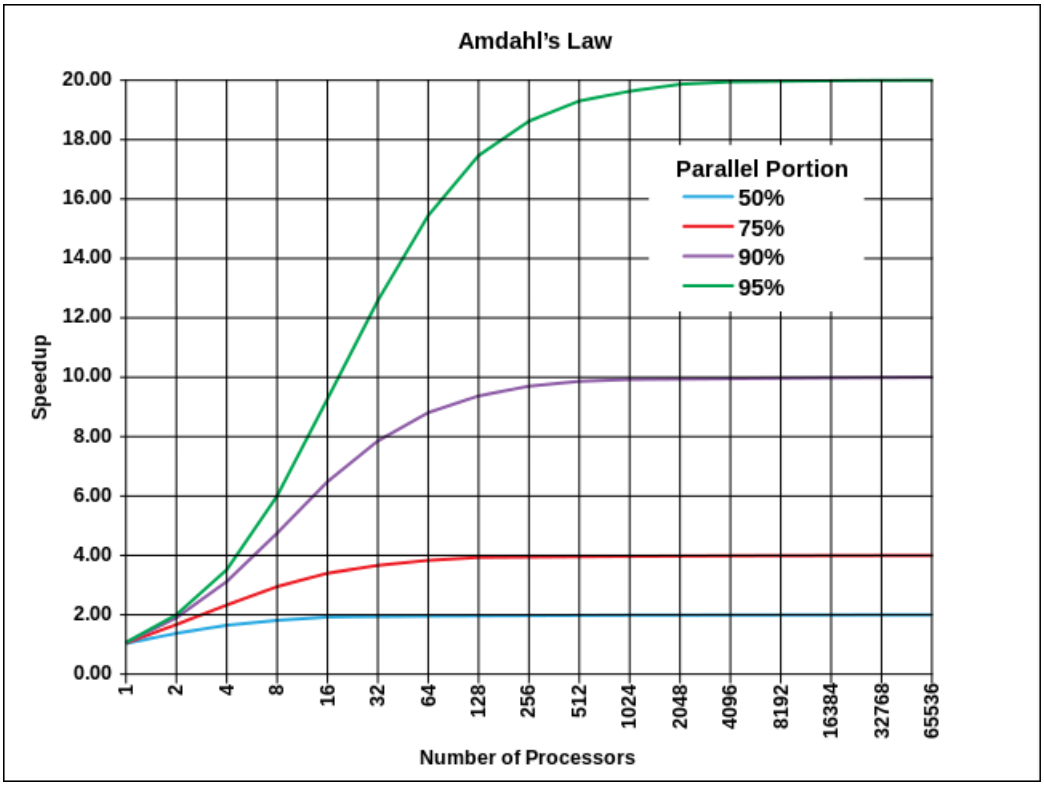

Fig. 1.1: Amdahl's Law, used via Wikimedia Commons. Used under the CC-By-SA 3.0 license.

Furthermore, while some code is easy to write as independent of other processes, one must eventually consolidate it. Imagine again that those trucks from the aforementioned shipping firm also had to pick up parcels from their destinations. This adds another layer

of complexity, the central truck command must now be there not just to hand instructions to the drivers but needs to be able to accept packages. Similarly, there must be some part of the computer that is dedicated to assigning tasks and consolidating results. It must also be on a process that is in charge of instantiating parameters and populating the computer memory, and must return some kind of indication to the user that everything is done. Worst of all, unless one is going to be in the very base operating kernel of the computer, this same process must also bear the burden of running the operating system and the overhead of the programming language that one is using. Essentially, the one who is doing the assigning of tasks is also the one who is running the entire world on which everything is based. So we can see that before we have really begun to give special instructions, one of our processors must be using some 12 to 20 percent of its real available resource (unless you are running the E series or the x99 series processors and or linux).

I again refer to the example above, to point out that trucks need homes to return to when they are finished with their tasks. Likewise, CPU's have a limited number of cores that they can physically house. The computer that I am using (for the bulk of this project) has four physical cores that are Hyper-Threaded and can be treated as 8 logical cores. I will not talk at length about threading here (Please see part 2), but we can treat this for now as a simple quad core machine. If I want to do more work with more cores, the simple solution is to simply pick up another machine and have it talk on a network to share information and resources. By definition this is a server. Thus we can have many machines that can communicate with one central process that handles all of the requests and consolidates all of the returned information. But the overhead in a system like this is somewhat large, and it is really annoying to set up and program (See part 2). It is like running Wal-Mart, we need trucks everywhere accross the world so we can service big distribution centers that in turn can dispatch their own trucks. This adds a rapidly growing heirchy of command as we now need a process to be in charge of the cluster of processes in charge of further processes.

Thus we went with one of the beautiful solutions.

1.5 The beautiful solution: GPU programming

Hooray for supercomputers, that is, hooray for computer clusters!

That is what I would like to say, but supercomputing time is hard to get and so we decided on a more practical approach that was easier to implement. We decided to do the bulk of our programming accross a GPU.

Imagin if you will that instead of picking up all and doing all sorts of things, we just needed to drop off and pick up postcards. Well, we don't need to have a large fleet of trucks, this is a waste of resources and they are cumbersome. Instead, we hire a massive fleet of people on scooters to pick up just a few things from their designated areas and bring it back right away. These are much lighter vehicles than a large semi and can be used for very simple tasks. This is the GPU.

The GPU is usually responsible for the video output of a computer. It treats a monitor screen as a massive matrix where it assigns values corresponding for every pixel on the screen. As an example, for the screen currently being used by myself as I write this thesis, the matrix corresponds to a 3840 x 2160 matrix. The GPU refreshes this matrix 75 times a second, in other words, it must change 8.3 billion values 75 times a second. It is able to accomplish this because it is highly parallel. It can treat each pixel independent of each the other making it an ideal tool for linear algebra.

The GPU also has a lot of processors in it. This particular machine has 1664 GPU core processors that shares 4 GB of GDDR5 ram. Each of these cores (CUDA cores) are less powerful than a CPU core but they are fine tuned to be able to handle linear algebra very well. For simple massive parallization, they are sufficent to the task.

This GPU also has a special feature that NVIDIA corporation has called the Maxwell microarchitecture. They have added 8 scheduler APU's to manage the information dissemination and collection amongst the CUDA cores, these APU's can issue two instructions that are independent simultaeneously. Because they are only nanometers apart from the CUDA cores, all the CPU has to do is communicate instruction to the GPU main processor which can relay the instruction to its APU schedulers. Thus the cumbersome task of que-

ing and assigning resources to the processes is all done outside of the CPU and is mostly independent of the operating kernel. From the CPU's perspective, all it has to do is send a block of vauge instructions that the GPU can understand and then interpret the reply that the GPU returns.

The use of the GPU has many challenges in coding and implementation, but this offers a very satisfactory boost in speed and reduces the overhead of many of the high level languages.

Chapter 2

Language Barriers

2.1 Initial Considerations and Benchmarks

It is probably readily apparent to the casual observer that the closer one can get to the hardware, the faster it is to send tasks and information. Thus it is with programming languages that the more basic and streamlined for the processor a language is, there is a speed difference. However, it is really difficult to talk in binary, or assembly which the transistor and the CPU are natively handeling. The lower the language, the less overhead we have but the harder it is to implement, debug and get into production/distribution. We can go with a good all-round lower level language like C++ which is fairly quick or with a higher level language like Python which is slow. There are also languages like VBA, Matlab and JAVA that we can also look at.

We first test the basic languages against each other using the standard Black-Schoals formula and see what speeds we see. We choose to program in the cumulative density function (although that is an available library in python) for testing purposes.

2.2 Black Scholes as a Test model

This is a very beautiful solution (closed form and analytical) to the heat diffusion equation. My understanding and simple derivation can be found in the appendix. We will be using this formula as a benchmark for most of the initial tests as well as using it for control variate variance reduction. Note that we will also be using MCMC for most of what we are testing. From my perspective, a random walk is probably not the best choice to exemplify the stochasticity of this particular field that we are applying these tools to, but a Levy flight is harder to easily implement.

2.3 Testing Serial Processes

To set our simple benchmarks, we choose to write code in three languages: C++, Python and Matlab. These languages were chosen because of two reasons, 1) I am very familiar with these three and 2) in relativistic mechanics, one can assume (until further information is acquired) that what we know and observe is the status quo for everythin, thus, I can assume that everyone is familiar-ish with perhaps one of these three.

We are going to implement a very simple script in serial. These will run on one process and will return 40000000 iterations of a black scholes pricing model. The excecution time of interest will be based on just the BlackScholes function loops, this timer will start when the function is called and end when it exits. Three timers will be used in total: Single instance (just one iteration), Stress instance ($40 * 10^6$ iterations) and the full program runtime from initialization to completion.

The hardware that we are going to be using to run our code is the following:

Component	Name	Speed/Size	Other
CPU	Intel i7-3770K	3.7 GHz	
RAM	Crucial	32 GB	DDR3 1600
PSU	EVGA Silver	850 W	Modular
Motherboard	ASUS Sabertooth z77		
GPU	EVGA GTX 970 Super-SuperClocked	1.317 GHz	1664 CUDA Cores

Table 2.1: Test Machine Specifications

The software that was chosen is as follows:

Language	Compiler / IDE	Additional Dependencies
C++	Visual Studio Professional 2013	AMP Included with Professional
Python	Pycharm	Anaconda (Python 2.7), Numba, Iopro
Matlab	Matlab 2014b	Finance, Parallel, Optimization (Toolkits)
OS	Microsoft Windows 7 Enterprise	

Table 2.2: Software and IDE

2.3.1 Simple Serial C++

C++ code excecution generally involves three steps. There is a preprocessor, a compiler

and a linker that handles the full compilation of the code. The preprocessor is designed to handle to preprocessor directives like include, define. It is outside of the syntaxical governance of c++ and so it can cause serious problems if misused, but this is a rare occurance unless one is really wading deeply in super-optimization of code. The preprocessor takes in tokens, declarations and produces a stream of tokens that are specially marked for the compilation engine to handle. Compilation is begun when the output stream of the preprocessor is recieved. This part of the code is all in c++, or at least should be. The compiler takes code and makes an object file, a file in binary format (machine language). The compiler does not care if symbols are not defined, it will happily (I am not actually sure if it is happy about any of this) take in and convert your code. However, syntax errors or overload errors will cause failures here.

Finally the linking mechanism takes the compiled output. It will link the symboles with addresses in your physical memory or to files in other libraries (include). If there are duplicate/missing definitions, the linker will complain. Thus in C++ (and C), there are three major types of issues that cause errors. Generally the errors will be in the compilation stage.

This is the C++ code, a little messy and we have to define everything. As you can see from the variable calls inside the BlackScholes function, we choose to send in a flag for puts or calls. Everything else we assume is best attributed by the double. the CND function merely takes in the d1 and d2 states of the BlackScholes function.

```
double BlackScholes(char flag, double S, double X, double T, double r, double v)
{
double d1, d2;
d1=(log(S/X)+(r+v*v/2)*T)/(v*sqrt(T));
d2=d1-v*sqrt(T);

// Price both the call and put
if(flag == 'c')
return S *CND(d1)-X * exp(-r*T)*CND(d2);
```

```
else if(flag == 'p')
return X * exp(-r * T) * CND(-d2) - S * CND(-d1);
}
// The cumulative normal distribution function
double CND( double X )
{
double L, K, w ;
double const a1 = 0.31938153, a2 = -0.356563782, a3 = 1.781477937;
double const a4 = -1.821255978, a5 = 1.330274429;

L = fabs(X);
K = 1.0 / (1.0 + 0.2316419 * L);
w = 1.0 - 1.0 / sqrt(2 * Pi) * exp(-L *L / 2) * (a1 * K + a2 * K *K + a3 *
    pow(K,3) + a4 * pow(K,4) + a5 * pow(K,5));

if (X < 0 ){
w= 1.0 - w;
}
return w;
}
```

2.3.2 Simple Serial Python

Python is a very happy language (to test this, in any python terminal type in: import antigravity). It is an interpreted language that defers nearly everything until runtime. Any variable or function are tagged when run with a type (int, str) and are pointed to. The values that are tagged are used to resolve what should be done, because of this 'Dynamic' style, the excecution is much slower as the computer must look at what the variable actually is before making a decision to proceed to the next line.

Python can be put to use in a wide sphere of applications and is platform independent making it a favorite of multi-system users like Google, Amazon and Server based processes.

This is a very pythonic code, the CND function is not shown. We pass in arguments and assign them to a variable. Python handles the memory allocation for us, by using the lists, this is slightly faster.

```
def BlackScholes(calllist, putlist, stockPrice, Strike, T,
                 Riskfree, Volatility):

    S, X, T, R, V = (stockPrice, Strike, T, Riskfree, Volatility)

    d1 = (np.log(S / X) + (R + 0.5 * V * V) * T) / (V * np.sqrt(T))
    d2 = d1 - V * np.sqrt(T)
    d1CND = CND(d1)
    d2CND = CND(d2)

    expRT = np.exp(- R * T)
    calllist[:] = (S * d1CND - X * expRT * d2CND)
    putlist[:] = (X * expRT * (1.0 - d2CND) - S * (1.0 - d1CND))
```

2.3.3 Simple Serial Matlab

We choose to simply use the built in toolkit for financial analysis. This is quick and simple but simultaensouly annoying. I don't like black boxes.

```
[Call, Put] = blsprice(Price, Strike, Rate, Time, Volatility, Yield)
```

2.3.4 Simple Serial Java

The Java code should look similar to that of C++, syntaxically these languages are very similar.

```
public double BlackScholes(char CallPutFlag, double S, double X, double T, double
    r, double v)
{
```

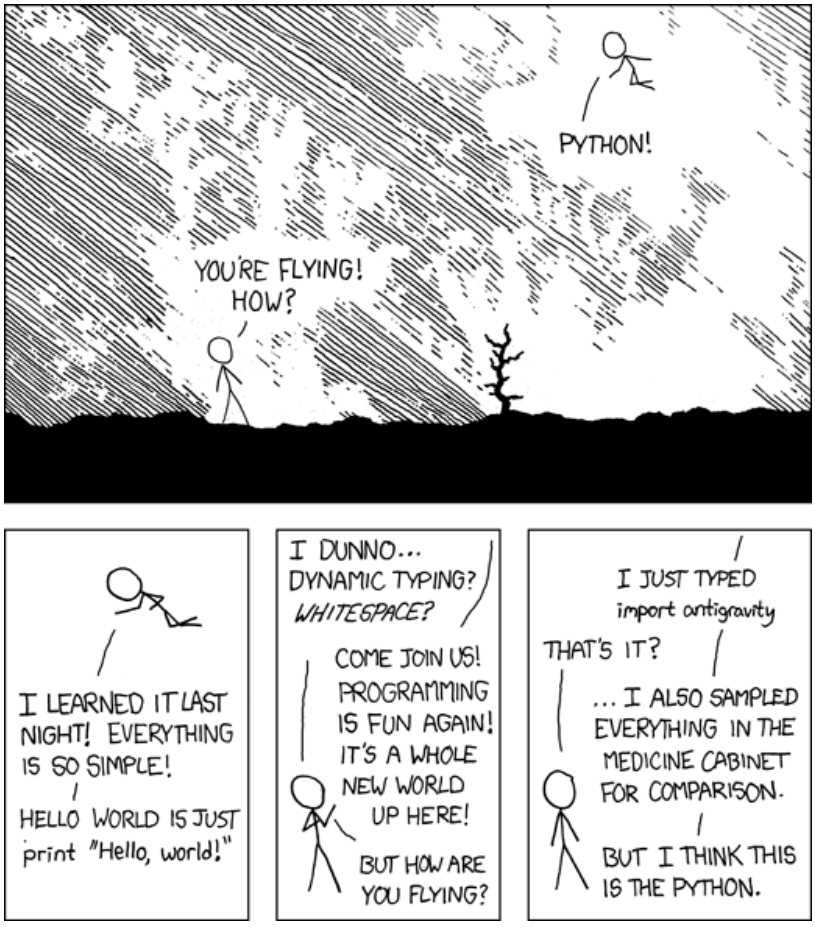

Fig. 2.1: Python: From XKCD http://xkcd.com/353/

```
double d1, d2;

d1=(Math.log(S/X)+(r+v*v/2)*T)/(v*Math.sqrt(T));
d2=d1-v*Math.sqrt(T);

if (CallPutFlag=='c')
{
return S*CND(d1)-X*Math.exp(-r*T)*CND(d2);
}
else
{
return X*Math.exp(-r*T)*CND(-d2)-S*CND(-d1);
}
}

// The cumulative normal distribution function
public double CND(double X)
{
double L, K, w ;
double a1 = 0.31938153, a2 = -0.356563782, a3 = 1.781477937, a4 = -1.821255978,
    a5 = 1.330274429;

L = Math.abs(X);
K = 1.0 / (1.0 + 0.2316419 * L);
w = 1.0 - 1.0 / Math.sqrt(2.0 * Math.PI) * Math.exp(-L *L / 2) * (a1 * K + a2 * K
    *K + a3
* Math.pow(K,3) + a4 * Math.pow(K,4) + a5 * Math.pow(K,5));

if (X < 0.0)
{
w= 1.0 - w;
}
```

```
return w;
}
```

We note that Java is a good all around language and uses a precompiler called the Java Virtual Machine to process its code which makes it independent of operating system platform (which is nice). However this sacrifices some speed, making the same speed as C/C++ virtually impossible to replicate.

2.3.5 Serial Task Conclusions

Below, we list the results of our initial benchmarks. The Matlab that we used is simply a built in function and is unkown how optimized it is. However, the matlab finance toolkit makes development much faster. Python is also very easy to develop in and was actually fairly optimized in just the simple script above (section 2.2.2).

Language	Time (single)	Time (multiple)	Time (total)	Development Time
C++	1 ms	2019 ms	2178 ms	20 minutes*
Python	0 ms	8314 ms	8529 ms	10 minutes*
Matlab	27 ms	1518 ms	1534 ms	
Java	2 ms	2831 ms	3026 ms	20 minutes*

Table 2.3: Serial Process Results

*These times are approximate

However we can see that in terms of sheer speed, the C++ is the clear winner for these serial tasks.

2.4 Multiprocessed Code

We will now jump into simple excecutions of highly parallel code. For this, we are going to show the extent of parralelization as well as the speed differences that we can achieve. The hardware and the software are the same, however we are going to be using C++ AMP, Iopro, and Parallel libraries for c++, python, and Matlab respectively. These libraries will allow us to communicate to the GPU and push the loops, and numerical processing to the GPU. This being the easily parrallelized components, we could probably optimize more of

the initial code as well as creating a threaded Queing algorithm, but that will be left as an excersize to the users.

We are going to be using Numba (Multiprocessing library for numpy) and Iopro (CUDA integration for Numba) which are both proprietary libraries from Continuum Analytics. Matlab and its toolkits (parallel) are proprietary libraries from Mathworks.

2.4.1 CPU only with Numba

In the initial test, we will use the above code from python and port it into using all of our existing cores (8 logical). We thank Continuum for their example with the CNDNumba function. We have to tell the processor server to listen for these functions, that is the real only difference between this code and the native python code [6]. We also have to separate the iterations a little into lists for multiprocessing, thus the loop in BlackScholes.

```
@autojit
def CNDNumba(d):
    A1, A2, A3, A4, A5 = (0.31938153, -0.356563782, 1.781477937, -1.821255978,
        1.330274429)
    RSQRT2PI = 0.39894228040143267793994605993438
    K = 1.0 / (1.0 + 0.2316419 * math.fabs(d))
    ret_val = (RSQRT2PI * math.exp(-0.5 * d * d) *
               (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))))
    if d > 0:
        ret_val = 1.0 - ret_val
    return ret_val

@autojit
def BlackScholes(calllist, putlist, stockPrice, Strike, T,
                 Riskfree, Volatility):

    S, X, T, R, V = (stockPrice, Strike, T, Riskfree, Volatility)
```

```
    for n in range(1, len(S)):
        d1 = (np.log(S[n] / X[n]) + (R + 0.5 * V * V) * T[n]) / (V * np.sqrt(T[n]))
        d2 = d1 - V * np.sqrt(T[n])
        d1CND = CNDNumba(d1)
        d2CND = CNDNumba(d2)

        expRT = np.exp(- R * T)
        calllist[n] = (S[n] * d1CND - X[n] * expRT * d2CND)
        putlist[n] = (X[n] * expRT * (1.0 - d2CND) - S[n] * (1.0 - d1CND))
```

2.4.2 CUDA with Python

We must first initialize the types of variables that this function will be taking. Cuda likes the C style, so we have a cythonic pre function part that does just that. Everything else should look very similar to the Numba code, however we are going to be calling the cuda library to pull the threads, blocks and the dim in the blocks. We do this to initialize the GPU and make sure that there is not a strange first term being qued in for processing. Note that this is using the iopro library (import numbapro, iopro, cuda). [2]

```
@cuda.jit(argtypes=(double[:], double[:], double[:], double[:], double[:],
                    double, double))
def black_scholes_cuda(callResult, putResult, S, X,
                       T, R, V):

    i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if i >= S.shape[0]:
        return
    d1 = (math.log(S[i] / X[i]) + (R + 0.5 * V * V) * T[i]) / (V *
        math.sqrt(T[i]))
    d2 = d1 - V * math.sqrt(T[i])
    d1CND = CNDCuda(d1)
    d2CND = CNDCuda(d2)
```

```
    expRT = math.exp((-1. * R) * T[i])
    callResult[i] = (S[i] * d1CND - X[i] * expRT * d2CND)
    putResult[i] = (X[i] * expRT * (1.0 - d2CND) - S[i] * (1.0 - d1CND))
```

2.4.3 CUDA with C++

We preface this by saying that C++ AMP has a really steep learning curve. This could be due to the fact that I am really bad at programming, but it took some time to get set up. (Please see my documentation and hints on my github repository: https://github/marioharper182/)

Here we have a BlackScholes already built that we are simple going to move the returned information from. We pass in the amp$_m athwhichistheparallelGPUmathlibrary.Wethenhavetoinitializeth$

```
#include "BlackScholes.h"
#include <math.h>
#include <amp_math.h>
#include <iostream>
#include <assert.h>

blackscholes::blackscholes(float _volatility, float _riskfreerate, int _size)
{
   data_size = _size;
   riskfreerate = _riskfreerate;
   volatility = _volatility;

   stock_price.resize(data_size);
   option_strike.resize(data_size);
   option_years.resize(data_size);
   call_result_amp.resize(data_size);
   put_result_amp.resize(data_size);
```

```
    srand(2014);

    for (int i = 0; i < data_size; i++)
    {
        stock_price[i] = 100.0f * (((float)rand()) / RAND_MAX);
    }

}
```

2.4.4 Limitations of Matlab

The GPU that I have is not optimized and prepared for Matlab parallel, while there are workarounds, I will have to wait to test until I have a workstation/supercomputing GPU that is already integrated with Matlab. I have a GTX series GPU which is better for all-around computing, it is too new for Matlab to be able to easily use.

We will put off tests until someone wants to buy us a TITAN or TITAN Black card.

2.4.5 Preliminary Multiprocessing Results

We can see that there is significant speed increases when we jump from 1 processor to 1664 processors. The first run overhead is higher as there is much more involved in getting the GPU initialized. However the results are satisfactory. The issue arises in the development time.

I believe that I can do this much faster now, but in terms of development and distribution, unless sheer speed is needed I would be loathe to recommend using the C++ Amp library. It is also somewhat dependent on the VisualStudio framework and works best with Windows and DirectX. The setup of the development environment is somewhat difficult as well.

2.5 Inherent Difficulties

In order to use the AMP library one must use VisualStudio, this means of course the

Language	Time (single)	Time (multiple)	Time (total)	Development Time
C++ Amp	47 ms	109 ms	149 ms	5 hours
Python CUDA	51 ms	127 ms	204 ms	2 hours
Python Numba	30 ms	427 ms	570 ms	1 minutes

Table 2.4: Multiprocessing Results

use of Windows OS. While Windows is a great operating system, Linux is much better. In terms of speed, linux is a faster alternative, but it requires near superhuman effort to load the GPU drivers (especially new GPU's) and coax the machine OS into making friends with new hardware. Because AMP is married so closely to the Windows environment we have to accept this OS overhead. We also acknowledge that DirectX is hard to work with. To get it to work we borrowed ideas from the UnrealEngine set up and it's DE environment.

Python has a lot more overhead as it is a higher level language. There does appear to be some abarrent memory bottlenecks with our bus because of it. Also, python is gluing together a lot of C code under the hood for us. While this is great, the extra layer of complexity means that we cannot match the speed of C based C++. Although we are using some proprietary CUDA libraries, they are open liscenced to education and dev use. They are also not as expensive as a Matlab toolkit even if purchase is required.

As can be seen below from the numpy array creation source code. We must first import python as a C library. The libary then pulls from other dependencies (ndim, idim, ect...) to asses the type of item it is and assign it a dimension. However, by doing this, we can see that the memory allocation issues from python are drastically cut.

```
#include <Python.h>


/* See array_assign.h for parameter documentation */
NPY_NO_EXPORT int
broadcast_strides(int ndim, npy_intp *shape,
                  int strides_ndim, npy_intp *strides_shape, npy_intp *strides,
                  char *strides_name,
                  npy_intp *out_strides)
```

```
{
    int idim, idim_start = ndim - strides_ndim;
    /* Can't broadcast to fewer dimensions */
    if (idim_start < 0) {
        goto broadcast_error;
    }
    /*
     * Process from the end to the start, so that 'strides' and 'out_strides'
     * can point to the same memory.
     */
    for (idim = ndim - 1; idim >= idim_start; --idim) {
        npy_intp strides_shape_value = strides_shape[idim - idim_start];
        /* If it doesn't have dimension one, it must match */
        if (strides_shape_value == 1) {
            out_strides[idim] = 0;
        }
        else if (strides_shape_value != shape[idim]) {
            goto broadcast_error;
        }
        else {
            out_strides[idim] = strides[idim - idim_start];
        }
    }
    /* New dimensions get a zero stride */
    for (idim = 0; idim < idim_start; ++idim) {
        out_strides[idim] = 0;
    }
    {
        npy_uintp start1 = 0, start2 = 0, end1 = 0, end2 = 0;

        get_array_memory_extents(arr1, &start1, &end1);
        get_array_memory_extents(arr2, &start2, &end2);
```

```
        return (start1 < end2) && (start2 < end1);
    }
    return 0;
```

Matlab is proprietary, wich means that it is going to cost some money to merely have a liscence of the IDE, not to mention the toolkits. The code is nicely packaged for us but the execution is a little slow. Not as many devices are supported and it does not play nice with the other kids. But it is platform independent, making it run natively on Unix, Linux and Windows.

Chapter 3

Lookback Call Option Under Stochastic Volatility

3.1 A Quick Introduction

This is a better example of testing the efficacy of programming languages as it is more involved of a process. In the fundamental sense, this is still a simple diffusion equation with a well defined boundary condition. Thus the equation governing the motion of each simulation follows the black scholes algorithm. We add the condition that the greatest value attained in a list of fixed points along a simulated path will be the value returned at the end of the iteration. We can express this as follows:

$$max(0, max(S_{terminali} : i = 1, ..., N) - K)$$

Where K is the value which represents a predetermined boundary value.

We also choose to make the assumption that the variance of the parameter V, ($V = \sigma^2$) is governed by the following differential equations:

$$dS = rSdt + \sigma Sdz_1$$

$$dV = \alpha(\bar{V} - V)dt + \zeta\sqrt{V}dz_2$$

We are also going to incorporate a known simple solution to a similar problem in order to compute the "greek" parameters. The simple solution has a continous fixed boundary value as well as a constant volatility. We show the analytical equation below [1]:

$$SimpleLookback = G + S\mathrm{e}^{-\delta T}N(x + \sigma\sqrt{T}) - K\mathrm{e}^{-rT}N(x)$$

$$-\frac{S}{B}(\mathrm{e}^{-rT}(\frac{\mathrm{E}}{\mathrm{S}})^{B}N(x + (1 - B)\sigma\sqrt{T})$$

$$-\mathrm{e}^{-\delta T} N(x + \sigma\sqrt{T}))$$

Where

$$B = \frac{2(r-\delta)}{\sigma^2}$$

$$x = \frac{\ln \frac{S}{E} + ((r-\delta) - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$$

3.2 The First approach

We first do a very simple approach to this problem via naive monte carlo. First we create a Weiner process and simulate a brownian path of length N, duplicate this M times (where M is the number of iterations we wish to run). We can see that this will create a M x N matrix where the rows of the matrix shows each individual simulation and the columns give the time horizon. We can then find the maximum value reached for every row M, compute the mean and discount it to get the price of the European Lookback call option under this first glance approach. The code is below:

```
def matrixengine(self, spot, rate, sigma, expiry, N, M, strike, sigma2):
    ### Calculate the trivial lookback option
    newdt = float(expiry)/float(N) # Get the dt for the Weiner process
    dW = np.sqrt(newdt)*np.random.normal(0,1,(M,N-1)) # Create the brownian
        motion
    W = np.cumsum(dW, axis=1) # Set up the Weiner Process as a Matrix
    time = np.linspace(0, expiry, N) # Set the discrete time space
    tempA = np.zeros((M,1)) # Create an initial zero vector for the first
        column
    Wnew = np.c_[tempA,W] # Append the Weiner matrix to the zeros
    tt = np.tile(np.array(time),(M,1)) # Create a matrix of time x M so we
        have time for every iteration

    ### Calculate the lookback option ###
```

```
assetpath = spot*np.exp((rate-.5*sigma2)*tt+sigma*Wnew) #European option
    stuff
max_vals = [max(assetpath[i]) for i in range(0 , len(assetpath))]
    #Calculate the maximum values reached
option_val = [max(i-strike,0) for i in np.array(max_vals)] #Apply the
    Payoff Function
call_val = np.mean(option_val) #Take the mean
present_val = np.exp(-rate*expiry)*call_val #Discount to present
print(present_val) #Show people what we get
```

As can perhaps be readily grasped, the matrix oriented language Matlab is better suited for this level of computation. We can also see that there is a speed advantage to Matlab as this is what it was designed to do. The code is also significantly shorter:

```
dW=sqrt(dt)*randn(M,n); % Generate array of brownian movement
W=cumsum(dW,2); % Obtain Wiener process
t=0:dt:T;
W=[zeros(M,1),W];
tt=repmat(t,M,1); % Create matrix with respect to time
asset_path=S*exp((r-0.5*sigma^2)*tt+sigma*W); %Calculate European stuff
max_vals=max(asset_path,[],2); %Get max values
option_values=max(max_vals-K,0); %Apply Payoff
present_vals=exp(-r*T)*option_values; %Discount
call_value=mean(present_vals);
display(call_value) %Show people what we get
```

3.2.1 The Control Variates

We choose to use the continuously fixed floating strike lookback call option formula to compute the delta, gamma and vega parameters. These will serve as the hedge control variates. This formula is valid for the case of constant volatility, and it will continuously

fix using the maximum value reached up to the point in time where the formula is at, ie:

$$time(t) = expiry - t$$

We create an greek parameters library file of functions that can be easily used to find the greeks:

```
def EuroDelta(d1):
    # Delta is the price sensitivity
    Delta = norm.cdf(d1)
    return Delta


def EuroGamma(d1, spot, sigma, tau):
    # Gamma is a second order time-price sensitivity
    Gamma = norm.ppf(norm.cdf(d1)) / (spot*sigma*np.sqrt(tau))
    return Gamma


def EuroTheta(d1, d2, rate, strike, tau, sigma, spot):
    # Theta is the time sensitivity
    Theta = -rate*strike*np.exp(-rate*tau) * norm.cdf(d2) - \
            (sigma*spot*norm.ppf(norm.cdf(d1))/(2*tau))
    return Theta


def EuroRho(d2, tau, strike, rate):
    # Rho is the interest rate sensitivity
    Rho = tau*strike*np.exp(-rate*tau) * norm.cdf(d2)
    return Rho


def EuroVega(d1, tau, spot):
    # Vega is a volatility sensitivity
    Vega = np.sqrt(tau)*spot*norm.cdf(d1)
    return Vega
```

```
def EuroD1(spot, strike, rate, dividend, sigma, tau):
    d1 = (np.log(spot/strike) + (rate - dividend + 1/2 *
                                 sigma**2)*tau) / (np.sqrt(tau))
    return d1


def EuroD2(d1, sigma, tau):
    d2 = d1 - sigma * np.sqrt(tau)
    return d2
```

The way that the control variates were implemented are a little messy. We break it into three basic sections: 1) Shorthand for all of the components that we will want to use with frequency later. 2) Initialization and processing of the control variates. 3) Implementing the Control Variate.

Part 1) We lay down the short-hand components with very unhelpful names, don't worry, we don't really want to know what they are doing under the hood of the engine anyways.

```
        # Calculate the parameter shorthand that we need for the control variates
        sig2 = self.sigma**2
        alphadt = self.alpha*self.dt
        xisdt = self.xi*np.sqrt(self.dt)
        erddt = np.exp((self.rate-self.dividend)*self.dt)
        egam1 = np.exp(2*(self.rate-self.dividend)*self.dt)
        egam2 = -2*erddt + 1
        eveg1 = np.exp(-self.alpha*self.dt)
        eveg2 = self.Vbar - self.Vbar*eveg1
```

Part 2) Call in all of the component parts that make up the hedging variates. We are going to be pulling in from the library we created earlier as well as liberally using numpy and scipy to get the distribution draws that we want.

```
        for j in range(1, self.M):
```

```
St1 = self.spot
St2 = self.spot
Vt = sig2

MaxSt1 = self.spot
MaxSt2 = self.spot
cv1, cv2, cv3 = 0,0,0

for i in range(1, self.N):

    # Initialize the d1, d2 variables for use in the control variates
    d1_St1 = EuroD1(St1, self.strike, self.rate, self.dividend,
        self.sigma, self.tau)
    d1_St2 = EuroD1(St2, self.strike, self.rate, self.dividend,
        self.sigma, self.tau)
    d2_St1 = EuroD2(d1_St1, self.sigma, self.tau)
    d2_St2 = EuroD2(d1_St2, self.sigma, self.tau)
    # These are the hedge sensitivites
    t = (i-1)*self.dt
    delta1 = (d1_St1)
    delta2 = (d1_St2)
    gamma1 = EuroGamma(d1_St1, MaxSt1, self.sigma, self.tau)
    gamma2 = EuroGamma(d1_St2, MaxSt2, self.sigma, self.tau)
    vega1 = EuroVega(d1_St1, self.tau, MaxSt1)
    vega2 = EuroVega(d1_St2, self.tau, MaxSt2)

    # Evolution of Variance
    e = np.random.normal(0,1)
    Vtn = Vt + alphadt*(self.Vbar - Vt) + xisdt*np.sqrt(Vt)*e

    # Evolution of Asset Price
```

```
            Stn1 = St1 * np.exp( (self.rate-self.dividend-.5*Vt)*self.dt +
                np.sqrt(Vt)*xisdt*e)
            Stn2 = St1 * np.exp( (self.rate-self.dividend-.5*Vt)*self.dt +
                np.sqrt(Vt)*xisdt*-e)

            # Process the Control Variates
            cv1 = cv1 + delta1*(Stn1 - St1*erddt) + delta2*(Stn2 - St2*erddt)
            cv2 = cv2 + gamma1*((Stn1-St1)**2 - St1**2 *
                (egam1*np.exp(Vt*self.dt)+egam2)) + \
                  gamma2*((Stn2-St2)**2 -
                      St2**2*(egam1*np.exp(Vt*self.dt)+egam2))
            cv3 = cv3 + vega1*((Vtn-Vt)-(Vt*eveg1+eveg2-Vt)) + \
                  vega2*((Vtn-Vt)-(Vt*eveg1+eveg2-Vt))

            Vt = Vtn
            St1 = Stn1
            St2 = Stn2

            if St1 >= MaxSt1: MaxSt1=St1
            if St2 >= MaxSt2: MaxSt2=St2

        CT = .5*(max(0,MaxSt1 - self.strike) + max(0, MaxSt2 - self.strike) +
                 self.beta1*cv1 + self.beta2*cv2 + self.beta3*cv3)
        sum_CT = sum_CT + CT
        sum_CT2 = sum_CT2 + CT*CT
        # print('Finished M loop')

    call_value = sum_CT/self.M *np.exp(-self.rate*self.expiry)
    SD = np.sqrt((sum_CT2 -
        sum_CT*sum_CT/self.M)*np.exp(-2*self.rate*self.expiry)/(self.M-1))
    SE = SD/np.sqrt(self.M)
```

Part 3) As we implement the actual method, it is simply a matter of making sure that the pieces that were instantiated in the previous lines are placed in the right order.

```
# Evolution of Asset Price
Stn1 = St1 * np.exp( (self.rate-self.dividend-.5*Vt)*self.dt +
    np.sqrt(Vt)*xisdt*e)
Stn2 = St1 * np.exp( (self.rate-self.dividend-.5*Vt)*self.dt +
    np.sqrt(Vt)*xisdt*-e)

# Process the Control Variates
cv1 = cv1 + delta1*(Stn1 - St1*erddt) + delta2*(Stn2 - St2*erddt)
cv2 = cv2 + gamma1*((Stn1-St1)^2 - St1^2 * (egam1*np.exp(Vt*self.dt)+egam2))
    + \
      gamma2*((Stn2-St2)^2 - St2^2*(egam1*np.exp(Vt*self.dt)+egam2))
cv3 = cv3 + vega1*((Vtn-Vt)-(Vt*eveg1+eveg2-Vt)) + \
      vega2*((Vtn-Vt)-(Vt*eveg1+eveg2-Vt))
```

3.2.2 Stochastic Volatility

We evolve the volatility over the life of the asset through the following mechanism:

```
    newdt = float(expiry)/float(N) # Get the dt for the Weiner process
    dW = np.sqrt(newdt)*np.random.normal(0,1,(M,N-1)) # Create the brownian
        motion
    W = np.cumsum(dW, axis=1) # Set up the Weiner Process as a Matrix
    time = np.linspace(0, expiry, N) # Set the discrete time space
    tempA = np.zeros((M,1)) # Create an initial zero vector for the first
        column

    #This is the Random aspects and the stochastic volatility
    Wnew = np.c_[tempA,W] # Append the Weiner matrix to the zeros vector
    Vt = self.sigma2
```

```
Vtn = np.abs(Vt + self.alphadt*(self.Vbar - Vt) +
    self.xisdt*np.sqrt(Vt)*Wnew)
tt = np.tile(np.array(time),(M,1)) # Create a matrix of time x M so we
    have time for every iteration

### Calculate the lookback option ###
assetpath1 = np.array(spot*np.exp((rate-.5*Vtn)*tt+np.sqrt(Vtn)*Wnew))
    #European standard
assetpath2 = np.array(spot*np.exp((rate-.5*Vtn)*tt+np.sqrt(Vtn)*-Wnew))
    #European standard
Vt = np.delete(np.c_[Vt+tempA, Vtn],-1,1)
```

3.3 The initial results:

We initially run a small set of 200,000 iterations and 10000 time steps of continous hedging, that is 20 Billion computations within the loops. The results can be seen in the table below:

	Naive	Control Variate
Time (ms)	412.100794134584	138241.19513435524
Standard Deviation	173.470605561235460	147.6637462506410
Standard Error	3.98741201421011351	3.406584621684048

Table 3.1: Simple Comparison of Control Variates

This table is quite a bit misleading as there is also a layer of complexity hidden by the stochastic volatility. In this case, initial volatility is set only to .09 and the stochastic factor is quite high for the case of the control variate. Thus it is actually very impressive that the control variate is able to have a lower standard error than a naive monte carlo.

3.4 Streamlining the Code

The biggest component slowing down the code is the inner loop where the control variates were being built. Because the control variates require the spot at every point in

time and the option is constantly hedging at every time interval, the code is required to query a series of external functions 20 Billion times in its run. It also must calculate each spot price and run a series of 18 calculations in sequence 20 Billion times, this is very slow.

Instead of having 200,000 simulations of a 10000 time step lookback option in a nested loop of i=200,000 and j = 10,000, we can treat this as a 200,000 x 10,000 matrix. In doing this, we can create a single object that we can perform operations on. While this may not sound like it saves much time, the results are listed on the table below:

	Naive	Control Variate
Time (ms)	262.000084	2064.999819
Standard Deviation	167.50330525143718	156.09686440621482
Standard Error	3.7454877699811102	3.4904319988686368

Table 3.2: Simple Comparison of Vectorized Control Variates

The code is also significantly shortened (some other smaller optimization was also performed for memory management but the table listed above is a result of only the vectorization). [5]

As can perhaps be seen, there are only MxN matrix operations being performed here. The math is altered slightly to incorporate this change to linear algebra.

```
# Initialize the matrices
newdt = float(expiry)/float(N) # Get the dt for the Weiner process
dW = np.sqrt(newdt)*np.random.normal(0,1,(M,N-1)) # Create the brownian motion
W = np.cumsum(dW, axis=1) # Set up the Weiner Process as a Matrix
time = np.linspace(0, expiry, N) # Set the discrete time space
tempA = np.zeros((M,1)) # Create an initial zero vector for the first column

#This is the Random aspects and the stochastic volatility
Wnew = np.c_[tempA,W] # Append the Weiner matrix to the zeros vector
Vt = self.sigma2
Vtn = np.abs(Vt + self.alphadt*(self.Vbar - Vt) + self.xisdt*np.sqrt(Vt)*Wnew)
tt = np.tile(np.array(time),(M,1)) # Create a matrix of time x M so we have
    time for every iteration
```

```
### Calculate the lookback option ###
assetpath1 = np.array(spot*np.exp((rate-.5*Vtn)*tt+np.sqrt(Vtn)*Wnew))
    #European standard
assetpath2 = np.array(spot*np.exp((rate-.5*Vtn)*tt+np.sqrt(Vtn)*-Wnew))
    #European standard

d1_St1 = EuroD1(assetpath1, strike, rate, self.dividend, sigma, self.tau)
d1_St2 = EuroD1(assetpath2, strike, rate, self.dividend, sigma, self.tau)
delta1 = (d1_St1)
delta2 = (d1_St2)
gamma1 = EuroGamma(d1_St1, assetpath1, sigma, self.tau)
gamma2 = EuroGamma(d1_St2, assetpath2, sigma, self.tau)
vega1 = EuroVega(d1_St1, self.tau, assetpath1)
vega2 = EuroVega(d1_St2, self.tau, assetpath2)

St1n = assetpath1
# St1n = np.c_[assetpath1, assetpath1[:,-1]]
St2n = assetpath2
# St2n = np.c_[assetpath2, assetpath2[:,-1]]
St1 = np.delete(np.c_[tempA, assetpath1],-1,1)
St2 = np.delete(np.c_[tempA, assetpath2],-1,1)

Vt = np.delete(np.c_[Vt+tempA, Vtn],-1,1)
# Vtn = np.c_[Vtn, Vtn[:, -1]]

cv1, cv2, cv3 = 0,0,0
# for i in range(len(St1n[0])):
cv1 = cv1 + delta1*(St1n - St1*self.erddt) + delta2*(St2n - St2*self.erddt)
cv2 = cv2 + gamma1*((St1n-St1)**2 - St1**2 *
    (self.egam1*np.exp(Vt*self.dt)+self.egam2)) + \
```

```
        gamma2*((St2n-St2)**2 -
            St2**2*(self.egam1*np.exp(Vt*self.dt)+self.egam2))
    cv3 = cv3 + vega1*((Vtn-Vt)-(Vt*self.eveg1+self.eveg2-Vt)) + \
        vega2*((Vtn-Vt)-(Vt*self.eveg1+self.eveg2-Vt))

    max_vals1 = np.array([max(assetpath1[i]) for i in range(0 , len(assetpath1))])
    max_vals2 = np.array([max(assetpath1[i]) for i in range(0 , len(assetpath1))])
    CT = .5*((max_vals1 - self.strike) + (max_vals2 - self.strike) +
            self.beta1*cv1[:,-1] + self.beta2*cv2[:,-1] + self.beta3*cv3[:,-1])
```

3.5 Multiprocessing with CUDA

The following code (as well as a couple of small structural changes) is added in order to have the gpu take the brunt of the calculations. We tell it to do a JIT (Just in Time) compiling, the target of the compuations is the maxwell gpu. The python code is going to be taking on very C like properties:

```
@cuda.jit(argtypes=(double[:], double, double, double, double, double[:],
    double[:], double, double, double,
                    double, double, double))
def VectorizedMonteCarlo(spot, rate, sigma, expiry, N, M, strike, Vbar, dt, xi,
    alpha, dividend, tau):

    i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
```

The resulting speedups in the same calculations are significant, as we now achieve 381.01795 ms in the Control Variate, stochasitc volatility case.

3.6 Conclusions

We have seen the following from our work on the European Fixed Strike Lookback Option in the table below. It is encouraging to see the speed differences from just the serial optimization.

	Naive	Control Variate
Simple	412.100794134584	138241.19513435524
Streamlined, Optimized	262.000084	2064.999819
CUDA	*Not Tested	381.01795

Table 3.3: Time Trial Results of European Lookback Option

From this we can see that there are real speed gains from the multiprocessing. However, in order to see more gains, we will have to write the code to be more paralilizable. Right now there is probably only 80 percent of the code that is able to be ported into a multiprocess. There are ways to increase this to 90 -99 percent but there is a real development time to realized gains tradeoff that we were unwilling to make.

Chapter 4

Other Examples

4.1 On Parasites

I choose to pull on some research that I have done to illustrate the point of GPU speeds in complex computations. In this example, we look at a parasite that is a helminthic infector. These are worms that have a complex life cycle and is a major public and health concern in parts of the world. Parasite lifecycle begins as eggs are transmitted to snails via mammal feces, the eggs hatch on the snail host and remain until the snail enters water. Once the snail is in water and the newly hatched parasites are well enough incubated, they turn into worms and leave the snail swimming up to the top of the water. These worms wait until a mammal comes by to get a drink, or for large mammals such as humans, to enter into the mammal through the pores of the skin. The parasite must then find its way to the liver of the mammal and there it stays until it can mate and pass on more eggs. The lifespan once situated can last several decades.

This is a slightly morbid example, but this illustrates the difficulty of modelling and constructing a computational approach. This is a highly nonlinear problem further exasperated by having drift terms within the water, the multitude of transmission vectors and the sheer amount of moving components. However, this is not an paper about parasite diffusion and control (please read my paper on that if you so wish) but on efficient computational methods. The simple code is as follows:

```
#Discretization of initial distribution
            x = []
            for i in range(0,self.np-1):
                newrange = -self.x1 + i*self.dx
                x.append(newrange)
```

```
x = array(x)
xlist.append(x)

I0 = array([0] * 255)

H0 = []
H0_element = array([abs(i) for i in x])
for i in H0_element:
    if i <=.25:
        H0_i = 20 * 1
        H0.append(H0_i)
    else:
        H0.append(0)
H0 = array(H0)

K = []
for i in x:
    equation =
        math.exp(-1*(i-self.v)**2/(4*self.D))/math.sqrt(4*math.pi*self.D)
    K.append(equation)

K_new = [i/(self.dx*sum(K)) for i in K]

fK = array(fft(K_new).T*self.dx)



for j in range(0, self.nsteps):

    fI0 = array(fft(I0))
    I0 = array(fftshift(ifft(fK*fI0)).real)
```

```
H = (1-self.mu)*H0+c*I0
I = (1-delta)*I0+self.b*((self.s-I0)*H0**2)/(1+H0)

for i in range(128, len(x)):
    if H[i] >= Hcrit:
        self.critvalue.append((i,j))
        self.critvalueH.append(j)
        self.critvalueX.append(i)
```

As can be readily seen, we must account for large matrices that have temporal and spatial components as well as a parasite density component. We take multiple fourier transforms of these matrices in order to approximate the equations of motion that governs the various aspects of time, space and density. (Incidently, this is not so different from the cannonical heat diffusion equation.)

4.2 On Asian Options

An asian option is another example of a path dependent option. It takes the average price accross its life as the final value. We can use control variates to pin down the price of the derivative with greater accuracy, of course this is a little expensive in terms of computing power. The GPU can speed up this process significantly.

```
def BlackScholesCall(S0, X, r, sigma, T, N, delta):
    d1 = (log(S0/X)+(r - deltas + .5*sigmas*sigmas)*T) / (sqrt(T)*sigmas)
    d2 = d1-sigmas*sqrt(T)
    Gtrue = ( S0*exp(- deltas * T)* norm.cdf(d1)) - (X*exp(-r*T)*norm.cdf(d2))
    return Gtrue
Gtrue = BlackScholesCall(S0, X, r, sigma, T, N, delta)

def CallPayOff(S, X):
    return np.maximum(S - X, 0.0)
```

```
def PutPayOff(S, X):
    return np.maximum(X - S, 0.0)

def MCAsianCall(S0, X, r, sigma, T, N, runs):
    Ct = np.zeros(runs)
    Ctg = np.zeros(runs)
    Cta = np.zeros(runs)
    St = np.zeros(N+1)
    nudt = (r - 0.5 * sigma * sigma) * dt
    sidt = sigma * sqrt(dt)


    for i in range(runs):
        z = norm.rvs(size=N)
        St[0] = S0
        for j in range(1, N+1):
            St[j] = St[j-1] * exp(nudt + sidt*z[j-1])

        #Do A + (Gtrue - Gsim)
        Ct[i] = CallPayOff(St[N], St.mean())
        Ctg[i] = CallPayOff(St[N], gmean(St))
        Cta[i] = Ct[i] + Gtrue - Ctg[i]

        #Add standard error, find the calculation in chapter 4
        linreg = reg(Ct[i],[Ctg[i]])
        SD = sqrt( (Ctg[i] - Ct[i]*Ct[i] / runs) * exp(-2*r*T) / (runs-1) )
        SE = SD / sqrt(runs)

    callPrc = Cta.mean() * exp(-r * T)
    print linreg, SE
    return callPrc
```

4.3 Results of GPU computing

We see that even in complex cases where not as much of the code is able to easily be ported into a parallel interface, we can realize the significant speed-ups.

	Serial	CUDA
Parasite Dispersal	6578.9810852096	1.008998716854
Asian Option	3.01984951654	.6401605467452

Table 4.1: Other examples of CUDA speedup

Chapter 5

Conclusions

There are some interesting findings in the work that was conducted. We first note that the speed we realize is largely dependent on how well we optimized the code before sending it into a GPU. We also find that Once written, multiprocessing accross a GPU with 1500 cores at its disposal largely nullifies the speed advantages between programming languages. There is inconsequential difference (depending on who you talk to) between the C++ AMP and NumbaPro Cuda libraries, but the development time overhead and bug fix is significantly higher in the C++ world. However, if one does not have access to a GPU of this spec, the results may vary. As a test, we have found that in just multiprocessing / multithreading tests using the CPU, C++ is vastly superior in performance.

With the seemingly simple implementation of GPU programming coupled with the vast speed increases that it provides, we cannot recommend the python-cuda approach more heavily. We suspect that Cython-Cuda may provide an even greater speed increase and will leave this as a future work.

References

[1] Les Clewlow and Chris Strickland. *Implementing Derivatives Models*. John Wiley Sons, West Sussex PO19 8SQ, England, 1998.

[2] Continuum Analytics. Introduction to python gpu programming with numba and numbapro, 2014.

[3] Ryan Garvey. Latency cost and information: Does speed matter for all market participants?residual vector quantizers with jointly optimized code books. *Journal of Trading*, 7:62–73, 2012.

[4] Robert L. McDonald. *Derivatives Markets*. Addison Wesley, 75 Arlington Streeet, Suite 300 Boston, MA, 1998.

[5] Stack Overflow - Christian Sarofeen. Cuda python gpu numbapro 3d loop poor performance, 2015.

[6] Stack Overflow - Joe. Python gpu programming, 2012.