GRAPHICAL ASIAN OPTIONS

MARK S. JOSHI

Abstract. We discuss the problem of pricing Asian options in Black–Scholes model using CUDA on a graphics processing unit. We survey some of the issues with GPU programming and discuss code design and memory usage. We show that by using a Quasi Monte Carlo simulation with a geometric Asian option as a control variate, it is possible to get prices that are accurate to 2E-4 within a fiftieth of a second.

1. Introduction

There has recently been much interest in using graphics cards to carry out computationally intensive tasks. Much of the work has focussed on the problem of pricing large numbers of simple contracts simultaneously with impressive speed-ups demonstrated. However, the problem a quant often faces is how to price one complicated contract quickly rather than how to price many straight-forward ones. In this note, we discuss the problem of pricing an arithmetic Asian option in the Black–Scholes model with time-dependent coefficients and show that speed-ups of the order of one hundred and fifty can be achieved by using NVIDIA's CUDA programming language on a graphics card. In particular, we will see that an option with 256 sampling dates can be accurately priced in approximately a fiftieth of a second. We will also discuss some of the intricacies and issues involved with CUDA programming.

The motivation for using graphics cards to price Asian options is that the GPU (graphics processing unit) is naturally parallel having been designed to carry out similar computations for every pixel on the screen. Given the "embarrassingly parallel" nature of Monte Carlo simulation, it becomes a natural candidate for pricing on the GPU. NVIDIA has launched the CUDA language which is an extension of C that allows GPU programming to be integrated naturally with CPU programs.

Whilst the commands for GPU programming are not hard to learn, programming requires different structure and thought patterns since the bottlenecks occur in very different places: floating point operations are fast, memory access is slow, and ifs can be slow. In fact, there are many different sorts of memory, and learning how and when to use each one is crucial for building fast models.

In this paper, we review the pricing of Asian options by Monte Carlo simulation. We then introduce the basic concepts of CUDA programming. Next, we decompose the problem of GPU implementation into five pieces:

- generation of Sobol variates;
- turning Sobol variates into normals;
- implementing the Brownian bridge;
- stock price path generation and pay-off computation;

Date: October 29, 2009.

Key words and phrases. CUDA, Asian options, GPU, Monte Carlo simulations.

- averaging across paths.

We finish with some numerical results.

2. Asian option pricing

In this section, we quickly review the pricing of Asian options by Monte Carlo. We refer the reader to [2] for detailed discussion. We assume that a dividend paying stock follows the process

$$dS_t = (r(t) - d(t))S_t dt + \sigma(t)S_t dW_t, \tag{2.1}$$

in the risk-neutral measure, with $r(t), d(t)$ and $\sigma(t)$ deterministic, well-behaved, function of time. We also assume the existence of a riskless bond B with

$$B_t = e^{\int_0^t r(s)ds}.$$

An arithmetic Asian option pays the average of the stock price across some pre-determined dates minus a strike floored at zero at some fixed time T. Thus if the sampling dates are $t_1, t_2, \ldots, t_N$, its pay-off is

$$\max\left(\frac{1}{N}\sum_{j=1}^{N} S_{t_j} - K, 0\right).$$

Its price using risk-neutral evaluation is

$$Z(T)\mathbb{E}\left(\max\left(\frac{1}{N}\sum_{j=1}^{N} S_{t_j} - K, 0\right)\right),$$

with S_t following the process above, and $Z(T) = \exp\left(-\int_0^T r(s)ds\right)$.

The price can be implemented using Monte Carlo simulation. Let t_0 be the current time. Let

$$\mu_j = \int_{t_{j-1}}^{t_j} (r(s) - d(s) - 0.5\sigma^2(s))ds$$

and

$$s_j = \sqrt{\int_{t_{j-1}}^{t_j} \sigma^2(s)ds}.$$

If W_j are a sequence of independent normals then we can simulate the stock price path by

$$S_{t_j} = S_{t_{j-1}} e^{\mu_j + s_j W_j}. \tag{2.2}$$

Once we have simulated the path, we compute the pay-off, discount it and average across many paths.

This method whilst correct can be slow to run and converge since there can be many sampling dates and the convergence of a Monte Carlo simulation is $\mathcal{O}(n^{-\frac{1}{2}})$, with n the number of paths. Various approaches have therefore been introduced to accelerate the pricing. Two standard ones are the use of geometric Asian option as a control variate, [4], and the use of quasi-Monte Carlo, or Sobol numbers, to increase convergence speed.

The idea of using a geometric Asian option is that its price is easily computed analytically since a product of jointly log-normal distributions is log-normal. Its pay-off is very similar to that of the arithmetic Asian option so the difference is of much lower variance than either individually. We therefore run a simulation to price the pay-off

$$\max\left(\frac{1}{N}\sum_{j=1}^{N} S_{t_j} - K, 0\right) - \max\left(\left(\prod_{j=1}^{N} S_{t_j}\right)^{1/N} - K, 0\right)$$

and add on the price of the geometric Asian option at the end. This method is known to be effective and the results of Kemna and Vorst, [4], suggest a reduction of variance by at least a factor of ten.

Quasi-Monte Carlo (QMC) is another standard technique for accelerating convergence instead of picking numbers randomly (or pseudo-randomly) we pick them in a such a way as to make them fill out space uniformly. Sobol numbers are probably the most-popular method of QMC in finance. They have the property of being good on low dimension but not so good in high dimensions.

We draw a vector of uniforms (u_j) from $[0,1]^N$ with N the dimension of the integral. In this case, N is the number of sampling dates.

These are converted to a vector of normals, (W_j), typically using an approximation to the inverse cumulative normal function. It is necessary to use (W_j) in such a way as to maximize the impact of the lowest dimensions. One solution is to use the *Brownian bridge* to construct the Brownian path. With this the first random number is used to determined the end-point of a path rather than the first increment. Successive random numbers are used to fill in the gaps. This ensures that the low dimensions have greater impact and has substantial effect on convergence speed. We therefore will use it when pricing. We refer the reader to Jäckel, [3], for detailed discussion.

Since these techniques have substantial effect when pricing on the CPU, it is natural to use them also on the GPU. Indeed, we will not gain a great deal if we shift to using a much slower to converge model even if each path can be run much more quickly.

3. The CUDA programming model

In this section, we discuss the basic ideas of CUDA programming on the GPU; it is inevitably not possible to discuss syntax in any detail. Instead, we will look at some of the differences with CPU programming. A CUDA program consists of two parts, the main routine located on the CPU and a number of *kernels* which run on the GPU.

When invoking a kernel, we have to specify how many *blocks* and how many *threads*. Each thread in each block runs independently. Threads within the same block can interact with each other, but different blocks cannot. A typical GPU can have up to 512 threads a block, and up to 65535 blocks. There is no guarantee as to order of execution; however, one can force all threads to wait until all have reached the same point in the code. This lack of ordering helps the GPU run faster in that when one batch of threads is being delayed by a memory access, another batch can still be processing.

Every thread will typically be executing identical code on different data. Branching conditional on data can therefore be a very slow operation, as half the threads wait around doing nothing whilst the other half executes the conditional branch. Threads are organized into batches of 32, and this is only a real issue when threads within a batch

diverge. One therefore has to either organize data so that each batch of threads does the same thing or write branch-free code. The latter appears to be generally easier.

Each thread has two identity numbers: the thread ID and the block ID. These are used by the thread to compute the location of the data to be processed. This allows the threads to carry out identical floating point operations on different data.

One of the biggest hurdles in writing an effective GPU program is understanding memory. There are, in fact, many different sorts of memory and each of this has different characteristics. The effective use of memory determines the speed of a GPU program. In particular, we have

- host memory,
- global memory,
- constant memory,
- shared memory.

Host memory is simply the ordinary memory of your computer controlled by the CPU. A kernel cannot access host memory, however.

Global memory is the main part of the memory on a graphics card with high-end cards having 750 MB to 4GB of memory. The main problem with it is that accessing it is slow and it is not cached. However, the GPU can *coalesce* memory accesses. Essentially, this means that if we have an address `start` and we make thread `X` access `start+X`, then many memory accesses happen simultaneously and take no more time than one thread accessing. An important part of code design therefore lies in ensuring that accesses to global memory are coalesced.

Constant memory is a small area of memory on the GPU which is read only for it. The CPU can, however, write to it. Constant memory is cached and therefore fast. It is an ideal place to store look-up tables which can be set up on the CPU and then copied across.

Shared memory is a small area of memory, e.g. 16k per block, on the GPU which can be accessed quickly. The threads within a block all use the same shared memory. This allows the possibility of communicating between threads.

Commands exist to copy from host memory to global and constant memory and back. These are slow, however. One therefore wishes a design that minimizes such copying.

Whilst all this memory allocation can sound fiddly, an open source project, Thrust, exists to make it easier. Thrust, [6], is similar to the standard template library (STL) and provides easy automation of many standard tasks. Just as with the STL if one wishes to create a vector one uses the `vector` template, with Thrust one has `host_vector` and `device_vector` to allocate memory for the CPU and GPU respectively. The classes automatically deallocate memory when the objects go out of scope, and all the natural copying and assignment operations work in the expected fashion. Once one has set-up all the data on the device ready for processing, one can obtain a raw pointer to its start and pass that pointer to a kernel.

Thrust also allows the calling of standard algorithms on both host and device data. Thus it is possible to apply the same function to all elements of an array on either the device or host, simply by calling `thrust::transform` algorithm within a CPU program. This gives a simple way to use the power of the GPU without having to do any real CUDA programming.

Another subtle aspect of GPU programming is that many graphics cards only support `floats` and not `doubles`. This is in contrast to typical derivatives pricing routines where `doubles` are standard. For Monte Carlo simulation, this is not a huge problem. Typically, the number of floating point operations per path is not large enough for a big floating point error to accumulate. Plus provided the errors for each path are independent of other paths, all they do is increase the standard error slightly and average out to a very small number. One has to be slightly careful about how one carries out the averaging in that as this point, a large number of floating point operations are involved. However, Thrust provides a `reduce` algorithm to do this for us. Or one can simply use double precision arithmetic on the CPU for the averaging.

The new Tesla cards do support double precision, however, it is much slower than single precision, eg a factor of 8. Thus even in that case it is desirable to use floats rather than doubles where possible.

4. Breaking down the problem

We can regard the pricing of an Asian option as a sequence of operations:

(1) generating Sobol numbers;
(2) converting to normals;
(3) Brownian bridging the normals;
(4) generating the stock price paths;
(5) computing the discounted pay-off of the Asian option and its control;
(6) averaging across all paths.

Typically, in a CPU program, we would carry out all of these (except the last where we would update a running sum instead) for one path, and then do them all for a second path, and so on.

In a GPU program, we do each stage for all paths before moving onto the next one. Thus we create the following kernels:

(1) a Sobol generator;
(2) an inverse cumulative normal transformation;
(3) a Brownian bridge;
(4) a stock price evolution and discounted pay-off computer;
(5) a reduction algorithm;

and each of these does all the paths simultaneously using the output of the previous step.

The first kernel is easy. We use Thrust to allocate device memory equal to the number of paths times the problem's dimension (that is the number of sampling dates.) We then use the Sobol implementation in the CUDA SDK provided by NVIDIA. After calling the kernel, we have all the uniforms stored in global memory on the device.

We do not copy them back to the host since there is no need; transferring data between host and device is probably the tightest bottleneck in GPU programming so we avoid it as much as possible. The output is done in a slightly strange way. We have dimension zero for all paths, followed by dimension one for all paths and so on. In CPU programming it is more natural to have all dimensions for path zero, and then all for path one and so on. We could use the transpose routine provided in the SDK to achieve this, but that

would be a mistake. This odd memory layout facilitates memory access coalescing and is therefore advantageous.

To convert all these uniforms to normals we could hand-code a kernel, however, the Thrust library's `transform` algorithm is sufficient. We code a simple function to do the inverse cumulative normal and then apply to all elements of the array. The only real issue here is that we want an inverse cumulative normal that runs fast on the device. This means no branching; we used an implementation due to Shaw and Brickman, [5].

Bridging is trickier; here we have to do some real work. Our objective is to create a bridge with $t_j = j$. We then take successive differences in order to obtain standard normal increments. Note that if we used truly random numbers this scrambling would have no effect on their distributions, but with quasi-randoms we should receive greater impact from low dimensions.

We use one thread per path. We require the dimension to be 2^n, increasing N as necessary. We divide the bridge into four phases:

- Determining the final point of the path,
- Filling in the mid-point of the left-most gap until we reach the first point, and the left-most gap is empty.
- Filling in the remaining gaps, starting with the largest ones, and then repeating until all gaps are filled.
- Taking successive differences to get standard normals.

The distinction between the second and third stages is that in the second stage the new point depends on one previously drawn point to its right, whilst in the third stage it depends on two: one on the left and one on the right.

To fill in gaps in the one-sided part, we need to know which index to do, which index is to the right, what multiple of the index to the right to use, which variate to use and what multiple of that variate is appropriate. For the two-sided part, we need this data plus analagous data for the left-side point. We compute all these arrays once only and then use them without change for every path.

Our approach is to create these arrays using the CPU and then copy them into constant memory on the GPU. We present the kernel code.

```
    __global__
void brownianBridgeGPU__kernel(float firstMidScale,
                               float* input_data,
                               float* output_data,
                               int PowerOfTwo,
                               int number_dimensions,
                               int number_paths)
{
 int bx = blockIdx.x;
 int tx = threadIdx.x;

 int gx = gridDim.x;
 int bwidth = blockDim.x;
 int width = gx*bwidth;

 int pathsPerThread = 1 + (( number_paths -1)/width);

 for (int l=0; l < pathsPerThread; ++l)
 {
```

```
    int pathNumber = width*l + bwidth*bx+tx;
    if (pathNumber < number_paths)
    {
      float* input_data_offset =  input_data + pathNumber;
      float* output_data_offset =  output_data + pathNumber;

// end-point of path
     *(output_data_offset+(number_dimensions-1)*number_paths)
                              = input_data_offset[0]*firstMidScale;

// one-sided bridging
     for (int i=0; i < PowerOfTwo; ++i)
     {
       int index = dev_const_indexToDoOneSided[i];
       float rscalar= dev_const_rightNotLeftScalars[i];
       float bv = output_data_offset[dev_const_indexRightOneSided[i]
                                                    *number_paths];
       float variate = input_data_offset[
                          dev_const_variateToUseForThisIndex[index]
                                                      *number_paths];
       float volscalar = dev_const_midNotLeftScalars[i];
       output_data_offset[index*number_paths]
                                          = rscalar* bv+ volscalar*variate;
     }

// two-sided bridging
     for (int i=0; i < number_dimensions- PowerOfTwo-1; ++i)
     {
        int index = dev_const_indexTodo[i];
        output_data_offset[index*number_paths] =
            dev_const_rightScalars[i]
              *output_data_offset[dev_const_indexRight[i]*number_paths]
            + dev_const_leftScalars[i]
              *output_data_offset[dev_const_indexLeft[i]*number_paths]
            + dev_const_midScalars[i]
              *input_data_offset[
                       dev_const_variateToUseForThisIndex[index]
                                                    *number_paths];
     }

 // successive differencing
     for (int i=1; i < number_dimensions; ++i)
     {
         int index = number_dimensions-i;
         output_data_offset[index*number_paths]
               -=      output_data_offset[(index-1)*number_paths];
     }



  }
 }
}
```

Note that all the arrays commencing `dev_const_` are held in constant memory and have already been initialized before the kernel is called.

There are some standard aspects to the routine. The global keyword indicates that we have a GPU kernel that can be called from the CPU. We need to know the identity number of the thread, the identity number of the block and how many blocks in the grid (i.e. how many blocks in total) in order to find the data relevant to this thread. Each thread may do more than one piece of data, depending on how many paths we wish to do since we have a ceiling on the total number of threads times blocks. The code is written to work for any number of threads per block, and blocks in the grid. The user can then try different combinations to see what is optimal. Typically using 64 threads per block tends to work well since this meshes well with optimal memory coalescing.

After doing the bridge, we have a matrix of standard normals to be plugged into a path-generator and then pay-offs are computed. We could decompose into two kernels: developing the paths and storing them, followed by pay-off computation. By not doing so, we avoid the necessity of storing the paths. Note, however, to develop a flexible routine we could store the paths, copy them to the CPU and then do pay-off computation on the CPU which would be fast since no complicated operations are required. We declare device vectors to store the pay-off results: `outputDataArithmetic` and `outputDataGeometric`.

```
__global__
void
AsianCallGPU__kernel(float* input_normals,
                     int totalPaths,
                     int stepsPerPath,
                     float logSpot0,
                     float df,
                     float strikeArithmetic,
                     float* outputDataArithmetic,
                     float strikeGeometric,
                     float* outputDataGeometric)
{
 int bx = blockIdx.x;
 int tx =  threadIdx.x;
 int gx = gridDim.x;
 int bwidth = blockDim.x;
 int width = gx*bwidth;

 int pathsPerThread = 1 + (( totalPaths -1)/width);

 for (int l=0; l < pathsPerThread; ++l)
 {
   int pathNumber = width*l + bwidth*bx+tx;
   if (pathNumber < totalPaths)
   {
     float* input_data_offset =  input_normals + pathNumber;

     float runningSum  = 0.0;
     float runningGeomSum = 0.0;

     float logSpot = logSpot0;

     for (int i=0; i < stepsPerPath; ++i)
     {
         logSpot  +=  input_data_offset[i*totalPaths]
                              *dev_const_logSds[i]
```

```
                                +dev_const_drifts[i];
            runningSum += exp(logSpot);
            runningGeomSum += logSpot;
        }

        float payOff = runningSum/stepsPerPath - strikeArithmetic;

        payOff = payOff >0 ? payOff  : 0.0f;

        *(outputDataArithmetic + pathNumber) = payOff*df;

        float gAverage = exp(runningGeomSum/stepsPerPath);

        float gPayOff = gAverage -  strikeGeometric;

        gPayOff = gPayOff >0 ? gPayOff  : 0.0f;

        *(outputDataGeometric + pathNumber) = gPayOff*df;
    }
  }
}
```

The main input data are the normals, the drift for each step and the standard deviation for each step. The normals have already been produced on the device by the Brownian bridge. The drifts and standard deviations are produced on the CPU and then copied into constant memory. We also need the final discount factor, the strikes and the initial value of the log of the spot price. Our kernel outputs the discounted pay-off for each path of both the arithmetic and geometric Asian options. The routine is similar to that for the Brownian bridge but simpler. Note again that increasing `tx` by one, increases the location of memory accessed by one. This is again to ensure that memory accesses are coalesced.

After executing the kernel, we have the discounted pay-off for each path for each of the arithmetic and geometric Asian options. We use `thrust::reduce` to sum the pay-offs, and divide by the number of path to get their price estimates. Finally, we subtract the price estimate of the geometric Asian from the arithmetic one, and add on the analytically computed geometric Asian price and we are done. Note that the analytical formula can be implemented on the CPU using double precision arithmetic since it is a straight-forward short computation.

r	0.05
d	0.03
Spot	100
Strike	100
Expiry	1

TABLE 1. Parameters for the Asian option pricing

Volatility	GPU price	CPU Price	Error
0.1	2.72500	2.72509	-9.69E-05
0.12	3.16142	3.16156	-1.40E-04
0.14	3.59929	3.59944	-1.57E-04
0.16	4.03813	4.03813	-4.63E-06
0.18	4.47730	4.47727	3.67E-05
0.2	4.91670	4.91664	6.09E-05
0.22	5.35614	5.35609	4.45E-05
0.24	5.79558	5.79553	5.48E-05
0.26	6.23484	6.23486	-1.92E-05
0.28	6.67391	6.67401	-9.49E-05

TABLE 2. Pricing results

5. Numerical results

We price an arithmetic Asian option with 256 sampling dates which are evenly spaced. We use the Black–Scholes model with constant parameters as in Table 1. Constant parameters are chosen only to make reporting easy. We run ten different levels of volatility. We use 32,768 Sobol paths. The graphics card is an NVIDIA Quadro FX4600. For the Brownian bridging and path generation, we use 512 blocks of 64 threads. For the other kernels, we let Thrust and the NVIDIA CUDA SDK decide.

Averaging over 100 evaluations, we found that the average time to price an Asian option is 0.018 seconds. We present accuracy results in Table 2. The CPU price is obtained using 2^{22} Sobol paths with a geometric Asian option as control variate. Double precision arithmetic in C++ is used for this price. We see that the GPU model is accurate to better than $2E-4$. This is more than good enough. In comparison, running a similar routine using single-threaded C++ code on the CPU (Quad-Core Xeon) took 2.7 seconds. We have achieved a speed-up of about 150 times.

6. Conclusion

GPU technology has rendered the Monte Carlo pricing of Asian options sufficiently fast that there is no longer any need for analytic approximations. Whilst the speed of computation has been enhanced by the use of a geometric Asian option as a control variate, the general approach is applicable to many path-dependent exotic derivatives. The cost of a CUDA-enabled graphics card is low in comparison to the speed ups available.

References

[1] C. Benneman, M.W. Beinker, D. Egloff, M. Gauckler, *Teraflops for Games and Derivatives Pricing,* Wilmott July–August 2008
[2] P. Glasserman, Monte Carlo Methods in Financial Engineering,
[3] P. Jäckel, Monte Carlo Methods in Finance
[4] A.G.Z. Kemna, A.C.F. Vorst, A Pricing Method for Options Based on Average Asset Values, Journal of Banking and Finance, 14, 113-129, (1990)
[5] William T. Shaw, N. Brickman, Differential Equations for Monte Carlo Recycling and a GPU-Optimized Normal Quantile, arXiv:0901.0638v3 [q-fin.CP] 15 Feb 2009
[6] Thrust: a Template Library for CUDA Applications, http://gpgpu.org/2009/05/31/thrust

Centre for Actuarial Studies, Department of Economics, University of Melbourne, Victoria 3010, Australia
E-mail address: mark@markjoshi.com