

Computational Finance Technical Example

Introduction

This purpose of this document is to suggest ways of porting standard quantitative financial codes to the ClearSpeed Advance™ accelerator card. The discussion is motivated with examples using the Black-Scholes pricing model and a variety of numerical methods to price vanilla options. Each example has an accompanying source code implementation.

Assumptions

- Familiarity with ClearSpeed products, in particular the Advance™ card.
- Familiarity with Cn code, CSAPI and the vector math library.
- Familiarity with quantitative finance concepts.

Table of contents

1	Overview of examples	3
1.1	Host-card communication	4
1.2	Utility files	4
2	Black-Scholes analytic pricing formula	5
2.1	Background	5
2.2	Initial code description	6
2.3	Parallelizing sequential code	7
2.4	Incorporating the vector math library	8
2.5	Performance and possible improvements	9
3	European option pricing via binomial method	11
3.1	Background	11
3.2	Initial code description	12
3.3	Parallelizing sequential code	13
3.4	Incorporating the vector math library	14
3.5	Performance and possible improvements	15
4	Asian option pricing via Monte Carlo	16
4.1	Background	16
4.2	Initial code description	17
4.3	Parallelizing sequential code	17
4.4	Incorporating the vector math library	18
4.5	Performance and possible improvements	18
5	Pricing American options via explicit finite differences	20
5.1	Background	20
5.2	Initial code description	21
5.3	Parallelizing sequential code	22
5.4	Incorporating the vector math library	22
5.5	Performance and possible improvements	23

6	Broadie-Glasserman random tree method	24
6.1	Background	24
6.2	Initial code description	25
6.3	Parallelizing sequential code	27
6.4	Incorporating the vector math library	28
6.5	Performance and possible improvements	28
7	Summary	30
8	Bibliography	31
Appendix A	Reduction methods	32
Appendix B	cs_vgaussian stack overflow	33
Revision history	34

1 Overview of examples

Each of the examples described in this document is driven from a host executable run from the command-line. Each host executable is built using makefile included in the example directory. A top-level makefile exists to build each of the examples and an associated library. Navigate to the finance examples install directory and then into **option-pricing**. Build the examples by typing:

```
csmake -f Makefile
```

for Windows, or for Linux

```
make -f Makefile
```

Each of the examples consists of a top-level driving routine that parses any command-line options supplied when the executable is run. Depending on the command-line options supplied, either a reference implementation, a mono implementation, a poly implementation or an optimized vector implementation will be run. Calling the executable with a `-h` option prints a help message with a list of command-line options applicable for the example executable.

Each of the examples provides a reference ANSI-C implementation, along with detailed explanation of how the reference code reflects the quoted pricing formula. The reference implementation is not optimized for any specific computer architecture, its purpose is to describe the algorithm for the pricing method.

The mono implementation runs solely on the MTAP processor and does not make use of the 96 processing elements (PEs). Consequently, mono implementations take approximately an order of magnitude longer to execute than the equivalent program executing on the host processor. The examples allow you to pass in command-line options to control the execution. See the examples' help message for details.

The poly implementation makes use of the 96 PEs available on each CSX600 chip. Unless otherwise noted, all examples utilize both CSX600 chips on the ClearSpeed Advance card.

The vector implementation not only utilizes the 96 PEs, but also uses the vector math library that has been optimized for the ClearSpeed CSX600 architecture.

The code was not profiled during the process of porting the code to the ClearSpeed Advance card. It is always recommended that you profile your application before starting the porting process. The ClearSpeed Visual Profiler documentation [\[1\]](#) contains details of how to profile your application code both on your host and on the ClearSpeed Advance card.

1.1 Host-card communication

Each of the examples contains code to drive the Advance card and run programs on the CSX600 processor. The host source code is written in C and uses the CSAPI to set up and run the CSX600 processor and control the data transfer to and from the board.

Each run is timed using a timer accurate to one millisecond. Runtimes may vary depending on the platform used and what other tasks the OS is running. For details on the CSAPI and Cn please see [\[2\]](#), [\[3\]](#) and [\[4\]](#). The performance figures listed within this document are for the following system:

Operating system	Windows XP
Processor	Intel Xeon 3.2 GHz, 2 Gb RAM
C compiler	Microsoft Visual Studio 2005
SDK/runtime	2.50
Advance™ card	X620

Unless otherwise noted, all of the examples presented here use both CSX processors on the Advance™ card.

1.2 Utility files

Files common to each of the examples are placed in the **Utilities** directory of the installation. These files include cross-platform timing functions, math functions and functions layered on top of CSAPI to drive the Advance card. Calling the makefile generates a library file that is linked in when building each of the examples.

2 Black-Scholes analytic pricing formula

2.1 Background

In 1973 Fischer Black and Myron Scholes published the development of the Black-Scholes (Black-Scholes-Merton) model. The assumptions underlying this model have proved hugely influential within the finance community and they still form the basis of many modern models. Based on these assumptions, Black, Scholes and Merton also derived a differential equation that describes the price of any derivative dependent on a stock.

The key assumption involved the risk-free rate of return of a portfolio consisting only of a long position on a stock and a number of short call options on that stock. To simplify, you buy a stock and simultaneously sell the option to buy that stock. In the Black-Scholes world, the value of such a portfolio (assuming that the portfolio can be instantaneously rebalanced) will change at the risk-free interest rate [5].

The full assumptions are outlined as follows.

1. The stock price follows the Ito process described by:

$$\Delta x = a(x, t)\Delta t + b(x, t)\varepsilon\sqrt{\Delta t}$$

Where ε is a random drawing from a standardized normal distribution.

2. The underlying securities are liquid and can be perfectly hedged.
3. No transaction costs are involved with buying or selling securities.
4. No dividends are paid during the lifetime of the option.
5. No arbitrage opportunities are available.
6. Trading is continuous.
7. Risk-free interest rate is constant over the lifetime of the derivative.

These assumptions led to the Black-Scholes-Merton differential equation, where the option price $V(r, S, \sigma, t)$ is written as:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} = rV$$

The authors also supplied analytic solutions to this differential for a European style derivatives may only be exercised at the termination date defined in the option contract.

$$call = S_0N(d_1) - Ke^{-rT}N(d_2)$$

$$put = Ke^{-rT}N(-d_2) - S_0N(-d_1)$$

Where

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(\frac{r + \sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(\frac{r - \sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$N(x)$ is the cumulative probability distribution function for a standardized normal distribution. In other words, it is the probability that a variable drawn from a normal distribution will be less than x .

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{x^2}{2}} dx$$

For the simple circumstances described (non-dividend-paying stock, and so on), the given analytic formulae can be used to provide a fair value for a **call** or **put** option.

2.2 Initial code description

The analytic pricing formulae are simple to translate into C code. Most standard implementations of the C math library will not provide an implementations of $N(x)$, the cumulative normal distribution. The approximation used for $N(x)$ can produce dramatically varying results towards the limits of the domain, but a widely used approximation comes from Abramowitz and Stegun [6] via Hull [5].

In this example, the number of options to evaluate can be set on the command-line. The default is to evaluate 100,000 options. An appropriately sized buffer is allocated for each parameter and filled with randomly generated values. Depending on the command-line options supplied, the parameters will be passed to a reference implementation running on the host or to the ClearSpeed Advance board for processing.

Since the evaluation of each option is entirely independent of every other, both CSX processors on the Advance card can be used to run 50,000 evaluations on each processor. Half of each input buffer is transferred to each CSX600 processor. The calling application code hides the data-transfer mechanism. The exact implementation can be found in `\utilities\host_run_example.c`

The code makes use of the ability to allocate memory in the CSX processor's memory space.

The code is listed in `blackscholes_analytic_mono.cn` and differs from the host implementation in the addition of volatile global variables that constitute the interface between the Advance card and the host. These symbols for variables can be programmatically interrogated using CSAPI calls. Once the symbol is found, data can be written to and read from these symbols. Note that in the Cn code we use global pointers that are seemingly never initialized. We rely on the host allocating memory and initializing the pointers to point at the memory block. This is equivalent to calling `malloc` in the Cn code, but more efficient. The Cn implementation of `malloc` actually calls back to the host to request the

allocation of memory (as it is the host processor that controls memory allocation) before returning control to the CSX processor.

2.3 Parallelizing sequential code

Although the code now runs on the Advance card, it does not use the SIMD capability of the ClearSpeed architecture. We now need to parallelize the code across the array of PEs. The easiest way to do this is to notice the data parallel aspect of this problem. We can execute the same instructions (call the same functions) on every PE, with different data sets on each PE. This way we perform 96 evaluations in parallel (per CSX600).

Exploiting parallelism

In order to execute function or arithmetic operations on each of the PEs, rather than the mono execution unit, simply alter the qualifier for each of the variables involved in the expression. Looking at the parallelized version, you can see that the functions are identical, except for poly keyword qualifying each of the appropriate variables and functions. You must also remember to include the appropriate poly header files.

Unrolling the loop

We are now executing the loop kernel 96 as many times per loop iteration. We must unroll the loop to reduce the number of times the kernel is executed. The most straightforward way to do this is to increment the loop index by `__NUM_PES__`, rather than by 1.

```
for( i = 0; i < MAX_ITERATIONS; i += __NUM_PES__ )
{
    // kernel...
}
```

Copying data onto the PEs

Data transferred from the host into the card's memory banks is not immediately accessible from the PE array. You must copy the data into poly memory space. Likewise, you must copy data from poly memory back to the mono memory before allowing the host to attempt to read it. The data copying can be achieved by using the `memcpym2p` function for mono to poly transfers, or `memcpy2m` for poly to mono transfers. The transfer of data from mono memory to poly can cause delays so always profile the code and attempt to minimize transfers.

```
...
src_addr = &sigma[0] + (penum+j*__NUM_PES__);
memcpym2p( &psigma, src_addr, sizeof(double) );
for( i =0; i<NUM_PASSES; i++)
{
    pvalue = BlackScholes(pCallPutFlag, pS, pX, pT, pr, psigma);
}
dst_addr = &value[0] + (penum+j*__NUM_PES__);
memcpy2m(dst_addr, &pvalue, sizeof(pvalue));
...
```

2.4 Incorporating the vector math library

The code is now parallelized across the PE array. However, better performance can be achieved by aggregating four operations into a single instruction issue, increasing the amount of parallelism. Performance gains in the range of 2-4x can be gained with this further vectorization. In practice, this is achieved by unrolling the inner loop. When evaluating the Black-Scholes pricing kernel this is easy, since each evaluation is independent of the others.

Using the vector math library

The vector math library (VML) provides optimized versions of standard libm maths functions. This means that certain range-checking and exception handling has been removed to allow maximum performance. The functions supplied also operate on chunks of four words at a time. In order to use these, you must alter the variable datatype from `poly double` to `__DVECTOR`. You must also alter each of the C operators (`*`, `+`, `-`, `/`) to the appropriate intrinsic, and use the VML functions for each of the maths functions. The following code snippet shows how the functions and operators must be changed.

The poly version,

```
w = 1.0 / sqrt(2.0 * PI) *
      exp(- (X*X) / 2) *
      (a1*K +
       a2*K*K +
       a3*powp(K,three) +
       a4*powp(K,four) +
       a5*powp(K,five));
```

The version using the vector math library and optimized math functions,

```
t = __cs_vmul_scalar(cs_exp(__cs_vmul_scalar(
    __cs_vmul(*X, *X), -0.5)), 0.3989422804);
powK2 = __cs_vmul(K, K);
powK3 = __cs_vmul(powK2, K);
powK4 = __cs_vmul(powK3, K);
powK5 = __cs_vmul(powK4, K);
w = __cs_vmul(t, __cs_vadd(__cs_vadd(
    __cs_vmul_scalar(K, a1), __cs_vmul_scalar(powK2, a2)),
    __cs_vadd(__cs_vmul_scalar(powK3, a3),
    __cs_vadd(__cs_vmul_scalar(powK4, a4),
    __cs_vmul_scalar(powK5, a5)))));
```

In future versions of the SDK compiler, the use of intrinsics won't be necessary. Instead, the appropriate C operators will be overloaded, allowing users to simply change the type declaration of variables from `poly double` to `__DVECTOR`.

Unrolling the loop for vectorization

Each PE now performs four evaluations per loop cycle. We have effectively unrolled the loop a further four times.

```
for( i = 0; i < MAX_ITERATIONS; i += __NUM_PES__ * 4)
{
    // kernel...
}
```

2.5 Performance and possible improvements

Runtime performance

Here we are looking at the most important indicator of performance; run time. In general, how many seconds it takes to calculate “N” samples is important, but other factors, such as power, capacity and scalability may also be significant.

You can experiment with running the `blackscholes_analytic` host executable. The version to run can be specified using the “-v:” parameter and the number of samples to evaluate can be set with “-s:”. Systems with more than one board can set the number of boards to use with “-b:”. For example, to run the “vector” version with 1,000,000 samples on 2 boards, call:

```
\>blackscholes_analytic -b:1 -v:vector -s:1000000
Black-Scholes Analytic value 0.010160
1000000 samples in 0.063000 secs
```

The version parameter accepts four values, “reference”, “mono”, “poly”, and “vector”, each corresponds to the appropriate version to run.

It is important to note the mono version simply runs the C code on the mono execution unit and therefore we see a three-hundredfold increase in the run time. This is not surprising; the mono execution unit is not designed for high performance arithmetic and runs around one-tenth of the clock speed of a modern processor.

```
\>blackscholes_analytic -v:reference -s:1000000
Black-Scholes Analytic value 0.010160
1000000 samples in 1.703000 secs
\>blackscholes_analytic -v:mono -s:1000000
Black-Scholes Analytic value 0.010160
1000000 samples in 388.18000 secs
```

The parallel nature of this problem means we can easily distribute the compute over two CSX600 processors and we can see that the poly version is already nearly as fast as the processor (and running at far lower power). Finally, exploiting the optimized maths libraries improves performance sixteen times. It could be expected that moving to vector types improve the performance by a factor of four, yet it actually causes a factor of 16 improvement. This improvement is in part due to using optimized versions of math functions. Functions such as `expp`, `logp` and `sinp` are functions from `libm` that work on poly data-types. The vector math library provides optimized versions that work on poly or `__DVECTOR` datatypes. The optimized versions have the “`cs_`” prefix.

Improvements

There are currently 6 individual calls to `memcpym2p`, copying a total of 6*8 bytes to each PE. Calling `memcpym2m` or `memcpym2p` has a setup overhead, so we should aim to reduce this. More importantly, we can exploit the multi-threaded nature of the array processor and overlap the compute and IO. This leads to a double-buffer approach. The outline of such an approach is detailed in the following code.

```
// prime the pipeline by fetching 8 words from input array
async_memcpy2p64( SEM_M2P, &input_words[active_buffer][0],
                 (mono void * poly) (((char*)input_words)+pe_offset) );
__sem_wait(SEM_M2P);
while(1){
// prefetch the next chunk
async_memcpy2p64(SEM_M2P, &buf[!active_buffer][0],
                 (mono void * poly) (((char*)input_words)+pe_offset+stride)
);

    __sem_wait(SEM_M2P);
    //
    // bulk of processing here
    //
    async_memcpy2m64(SEM_P2M, (mono void *
                             poly) (((char*)results)+pe_offset),
                    &buf[active_buffer][0] );
    __sem_wait(SEM_P2M);

    active_buffer = !active_buffer;
}
```

After introducing this approach, the I/O time should be completely overlapped with compute and so the run time will only reflect the compute time of the algorithm.

3 European option pricing via binomial method

3.1 Background

Whilst the Black-Scholes-Merton pricing formula was an important milestone in options pricing, its applicability was limited. Particularly because it is not generally possible to accurately price early-exercise options such as American and Bermudan (in specific circumstances such as continuously-dividend-paying option it is possible to find closed-form solutions). In 1979, Cox, Ross and Rubenstein published a paper [7] applying the numerical method of Binomial Trees to pricing derivatives. They considered the pricing of an option on a non-dividend-paying stock. Assuming the option lifetime is divided into small time intervals, and at each time interval the stock value S can increase to uS with probability p , or decrease to dS with probability $q = 1-p$. Coefficients u and d are chosen to be recombining such that $Sud == Sdu == S$.

The expected value after one time step is:

$$(1) \quad pSu + (1-p)Sd$$

The expected variance of the return over Δt :

$$(2) \quad pu^2 + (1-p)d^2 - [pu + (1-p)d]^2$$

Setting (1) equal to the risk-free rate of return ($Se^{r\Delta t}$), and (2) equal to $\sigma^2\Delta t$, and with rearranging the algebra, we find:

$$p = \frac{e^{r\Delta t} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t}}$$

$$d = e^{-\sigma\sqrt{t}}$$

A European option can be priced using the following process.

1. Generate a tree of depth n , with the underlying stock prices at each node.
2. At expiry time, in other words at the leaves of the tree, optimally exercise the option at time $T (=t_n)$.
3. Step backwards in time, using the option value at t_{i+1} to find the value at t_i .
4. Repeat until the option price at $t_{i=0}$ is found.

It is apparent that a similar algorithm can be used to price early exercise options. At each time-step we check whether it is optimal to exercise at that time.

Note: The definitions of u and d create a recombining tree. Recombining trees have helpful properties in that it is simple to calculate the price at each node and there are $n+1$ nodes at depth n rather than 2^n nodes. The depth of the tree can be chosen according to the required accuracy of the pricing. As the number of steps approaches infinity, the value converges to the analytic price.

3.2 Initial code description

When generating the pricing tree, it is unnecessary to simultaneously hold all the values in memory. By working on a single stripe at any time step, the amount of memory required can be reduced to just the applicable “number of leaves”.

To simplify the code, allocate a maximum possible array size at compile time. We can check at run time that the requested number of time steps does not exceed this.

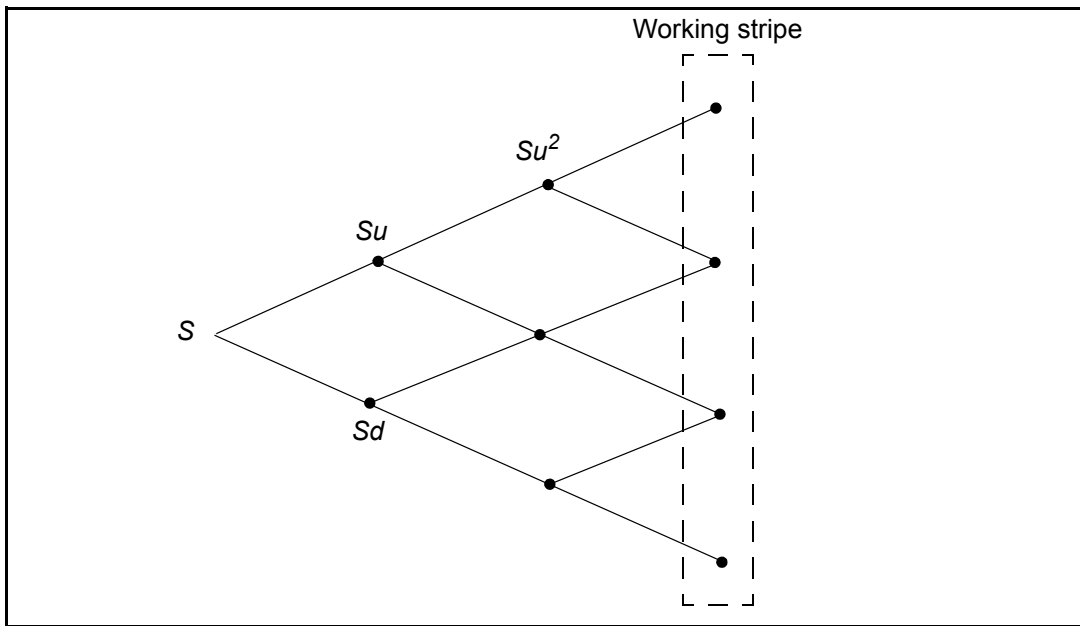


Figure 1. Tree configuration

The first two loops set up the underlying price at the leaves of the tree, and then fill in the call payoff for each of the prices. Once the boundary conditions contain the expiry date, the algorithm steps backwards in time updating the price array, holding the possible payoff values. As the time regresses to $t=0$, the width of the tree narrows. At time $t=0$ the result is held in `call_values[0]`.

3.3 Parallelizing sequential code

The array of `prices` and `call_values` is blocked onto the PE array.

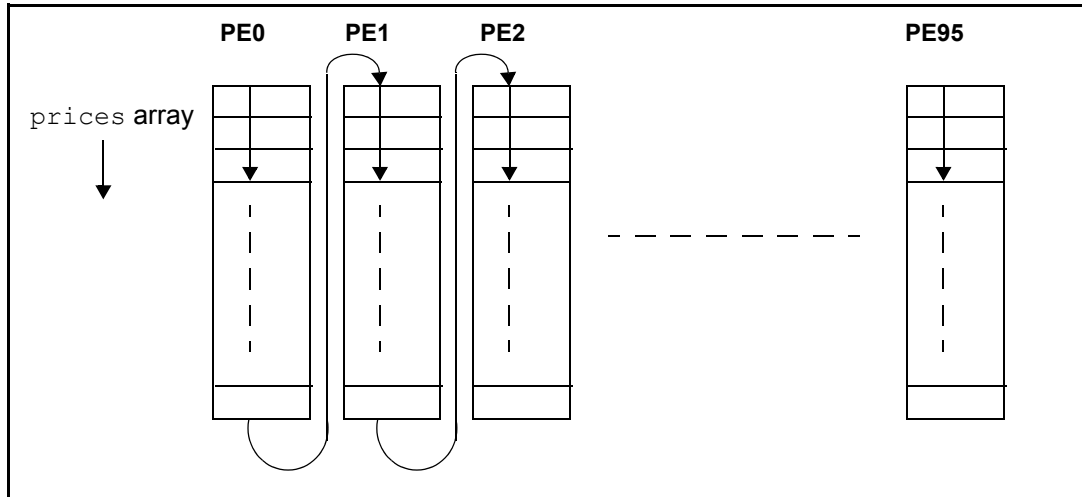


Figure 2. Blocking the array of prices onto the PE array

The data dependency introduced by the code:

```

for (i=0; i<=step; ++i)
{ // value of call_values[i] dependent on call_values[i+1]
  call_values[i] = (p_up*call_values[i+1] +
  p_down*call_values[i])*Rinv;
  prices[i] = d*prices[i+1];
  // ... exercise payoff
}
    
```

means that the data must move across the PE array:

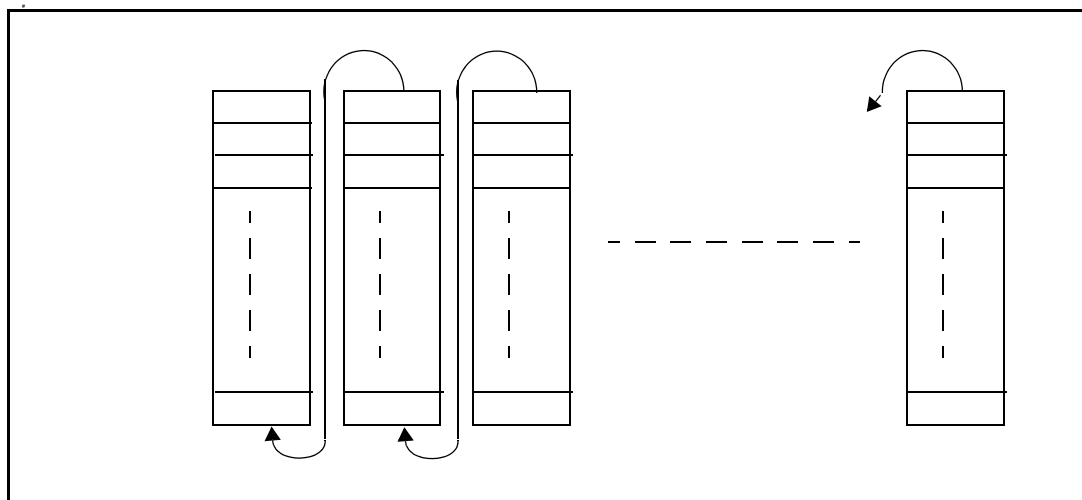


Figure 3. Moving data across the array

This is achieved using “swizzle” functions. Swizzling transfers data between adjacent PEs, so a `swizzle_down_double` call transfers a double word from PE `n` to PE `n-1`.

The basic parallelism strategy is intuitive and the most efficient. It is possible to consider other strategies that incorporate pipelining between PE elements but these require moving $O(n)$ data between each PE.

There are other possibilities to consider when laying the data across the PEs. For instance, keeping data local to PEs rather than striping it across the PE array. For large trees this is effective because when processing the leaves of the tree, local data can be worked on without the need for swizzling. However, there are several arguments against this data-layout tactic. Firstly, data will always need to be communicated through the PEs at some point in tree processing. So whilst initially forgoing swazzles, the eventual swazzling of data means marginally more complicated code. Secondly, typical tree sizes are not large enough to warrant this tactic.

3.4 Incorporating the vector math library

In order to make use of the vector instructions, the data layout must be altered again. Making best use of the vector instructions means that there can not be any data dependencies within a variable of vector datatype. The data can, therefore, be laid out as in [Figure 4: Moving data across the array](#).

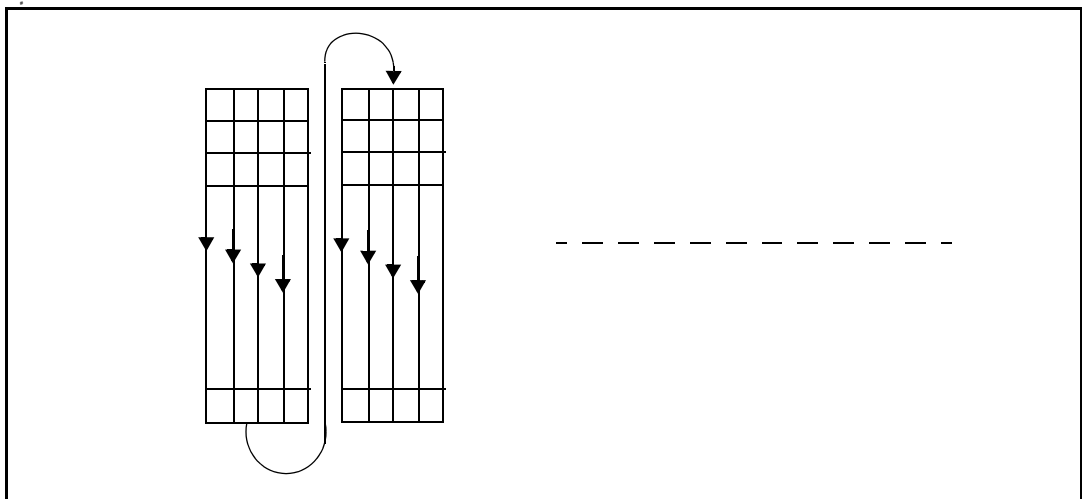


Figure 4. Moving data across the array

Where the data in the first element of the price array consists of the tuple $\{0, \text{chunk_size}, \text{chunk_size} * 2, \text{chunk_size} * 3, \text{chunk_size} * 4\}$ and so on. The result ends up in the first vector element of the first element of the `call_values` array.

3.5 Performance and possible improvements

Performance

Performance of the binomial tree method is bounded by the data transfer of input parameters to the Advance card. However, as the number of steps is increased (increasing the amount of computation relative to the initial data transfer) the performance improves.

It is worth noting that only one of the CSX600 processors on the Advance card is used in this example. In theory the problem is parallelizable across multiple chips (or even multiple cards). However, in practice, data-coherency issues associated with spreading the tree over multiple chips may mean that it is not efficient. The binomial tree can be sliced across multiple processors, but data will always need to be exchanged across the boundary. For processors on the same Advance card, memory can be shared, but this is complex. For CSX600 processors on different cards, exchanging data (across a PCI-X or PCIe bus) can be a costly operation. An alternative way to use both CSX600 processors and double the performance, would be to run two separate problems at the same time, one on each CSX600.

After running the executable, you should see the following output.

```
\>euro_binomial -v:reference
European Put (Binomial Method) value: 1.030803
European Put (Binomial Method) runtime: 0.875000secs
\>euro_binomial -v:vector -b:1
(1 of 2 available chips will be used)
European Put (Binomial Method) value: 1.030803
European Put (Binomial Method) runtime: 0.328000secs
```

This corresponds to a x2.7 acceleration. However, this acceleration depends on the number of on the number of steps in the binomial tree. Increasing the number of steps in the tree increases the accuracy of the option calculation, converging to the Black-Scholes analytic price. As the number of steps in the tree is increased, the data transfer time becomes a smaller proportion of the run time. For vanilla options in the Black-Scholes world there is little need for very fine-grain trees, but for complicated models, options or calculating the "Greeks", it can be necessary to use more steps.

4 Asian option pricing via Monte Carlo

4.1 Background

An Asian option has a payoff that is a function of the average of the underlying stock over some specified period of time. If the average is geometric and the stock follows a lognormal process, analytic solutions can be found, since the geometric average of N lognormal values is itself lognormal. However, if the average is arithmetic such solutions can not be found and numerical approximations must be used. A commonly used numerical technique is Monte-Carlo integration (or simulation). Simulation is conceptually very simple; it involves drawing a series of samples randomly from a particular distribution. In the limit, the arithmetic average converges to the integral.

Imagine a function $f(x)$, to be integrated over the range $[0,1)$.

$$(1) \quad I = \int_0^1 f(x) dx$$

If we approximate the integral as $E[f(\mathbf{U})]$, we can write an equivalent Monte Carlo estimate.

$$(2) \quad E[f(\mathbf{U})] = \frac{\sum_{i=1}^N f(u_i)}{N}$$

where \mathbf{U} is a set of uniformly distributed random numbers.

The error of the estimate is normally distributed and thus the error diminishes as $O(N^{-1/2})$. To simplify, increasing the quantity of random numbers used improves the estimate. Adding an extra decimal place of accuracy requires 100 times as many points. There are many techniques for improving the convergence of the error bound of the estimate, including antithetic variables, control variates, and stratified sampling [8].

Pricing of an arithmetically averaged Asian option can be helped significantly by the addition of a control variate. Assume derivative A is similar to derivative B, but an analytic solution is available for B. Both derivatives can be simulated in parallel using the same random number stream; the resultant estimate for A can be improved upon by adding the error between the different prices of B. In this example, control variate is the analytically determined geometric average Asian option price.

In general, running a Monte-Carlo simulation to price an option is simple.

1. Generate a uniformly distributed random value, u .
2. The underlying stock price is a stochastic variable and its variation is lognormal. Therefore transform the uniform variate to a normally distributed variate (with mean = 0.0 and standard deviation = 1.0).
3. Use the normal variate in the discrete SDE, and apply the payoff to give a value.
4. Accumulate the value. Repeat steps 1. to 4. for a suitably large number, N .
5. Scale the accumulated values and apply the discount factor.

Steps 1. to 4. constitute the simulation kernel.

With the addition of path dependency, Asian options can be priced using the same framework. In order to generate the average, sample the stock's path over M points in time.

4.2 Initial code description

Unique RNG seeds

It is desirable to have each simulation path use a unique set of numbers in order to ensure that no bias is introduced into the Monte Carlo estimate. For this purpose, each chip uses a different seed for `srand48(. . .)`. It is possible to find seeds that will ensure unique values are generated over the entire simulation, but we use the approximation is used in [9].

Generating paths

The inner loop generates the M-point path for each of the N paths simulated. The inner loop simulates both the geometric and the arithmetic price path evolution. The geometric average is defined as

$$\sqrt[M]{\prod_i S(i)}$$

To remove the need for an Mth root calculation, accumulate the logarithm of the stock price path instead.

As well as accumulating the call payoff, we also accumulate the square of the payoff to enable an estimate of the error bound of the simulation. The actual computation of the confidence level⁽¹⁾ is best executed on the host, rather than on the slow mono unit of the CSX600.

4.3 Parallelizing sequential code

The simplest strategy for parallelizing the code is also the most efficient. In this Monte-Carlo simulation each individual simulation is independent of the others. Therefore, we can perform the simulation kernel on each PE simultaneously, so that each PE performs a number of simulations. In order to do this, the functions called on each PE must be converted to poly functions.

Result reduction

The simulation loop accumulates results on each PE. At the end of the loop, 96 sub-results remain. In order to obtain a final result, the sub-results must be reduced to a single result. There are many ways to do this, but an efficient method is to use the swizzle path and accumulate data in mono memory as it is moved sideways. Once the fully accumulated result is in mono memory it can be transferred to the host. See Appendix A for more on reduction methods.

1. The confidence level is chosen as a representation of the error on the Monte Carlo simulation,

4.4 Incorporating the vector math library

The code is now parallelized across the PE array. However, better performance can be achieved by aggregating four operations into a single instruction issue, increasing the amount of parallelism. In practice, this can be achieved by further unrolling the inner loop. For a Monte-Carlo simulation this is simple as each evaluation is independent of the others.

Using the vector math library

The VML provides functions that operate on chunks of four words at a time. In order to use these, the variable data-type must be altered from `poly double` to `__DVECTOR`. Each of the C operators (`*`, `+`, `-`, `/`) must also be altered to the appropriate intrinsic, and use the VML functions for each of the math functions.

Unrolling the loop

As long as the number of simulation paths is further divisible by two, the outer loop can be further unrolled. The random number generator library supplies the function `cs_vdgaussian(...)` which generates two normally distributed random variates via the Box-Muller method.

Note: In SDK 2.50 `cs_vgaussian(...)` the stack can overflow. See Appendix B for details.

The last statement in the simulation kernel reduces the two results for this individual simulation from a `__DVECTOR` to a simple poly variable. It is this poly variable that keeps the running total over the simulation loop.

4.5 Performance and possible improvements

Performance

The performance of Monte-Carlo examples again shows good improvement once fully converted to use the high performance vector math libraries. The host program also prints a confidence level associated with the final answer.

The values printed for each version may not be exactly the same. These discrepancies arise from the differences in random number generators. This can be tested by overriding `drand48()` to return a constant in the range `[0.0,1.0)`. The discrepancies should not be large, but if further accuracy is needed, simply increase the number of simulations. As previously stated, to improve the accuracy by one decimal place requires 100 times as many simulations. The quality of random numbers produced by `rand48` is not high, so it may be preferable to use a Mersenne Twister function. ClearSpeed provide such a function, see [\[3\]](#) for details. Alternatively, generate random variates on the host and stream them across to the Advance card.

```
\>asian_mc -b:1 -v:vector
Confidence interval with control variate = (5.378995, 5.379009)
Asian Call (Monte-Carlo Method) value: 5.379002
Asian Call (Monte-Carlo Method) runtime: 0.219000secs
```

The vector version shows an increase of speed of approximately 30 times over the host version. Due to the nature of this algorithm, this scales perfectly onto multiple cards.

Improvements

The current version of the SDK does little to reorder instructions, minimize load/stores or optimize register spilling. As a result it is straight forward to find simple performance improvements using the ClearSpeed Visual Profiler or by looking at the assembler produced (use the `-s` option at the command line). It is worth trying to keep data in registers as long as possible. The following code leads to excessive load/stores:

```
poly double e1, e2, S, X1, X2
...
e1 = log(S);
e2 = log(S);
X1 = max(0.0, S-e1);
X2 = max(0.0, S-e2);
...
```

Rearranging the code can help to minimize the number of load/store instructions emitted by the compiler (future releases of the SDK will improve this).

ClearSpeed supplies a random number library with various common PRNGs, including `rand48`, `mcg59` and Mersenne Twister 19937. For further details of PRNGs provided by ClearSpeed, see [\[2\]](#) and [\[3\]](#).

5 Pricing American options via explicit finite differences

5.1 Background

Previous examples in this document have used the Black-Scholes model to find a numerical solution for pricing an option. Another obvious approach is to solve the partial differential equation itself. Solving PDEs has been the topic of research for many years in many disciplines, including fluid dynamics and structural mechanics, so many techniques are available. Please refer to the large quantity of literature available, particularly in the field of quantitative finance. An excellent introduction is found in [10], extensively examined in [11].

We want to solve the Black-Scholes differential equation

$$\frac{\partial V}{\partial t} + rS \frac{\partial V}{\partial S} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV$$

At time $t=0$, discretize asset price and time onto a Cartesian grid.

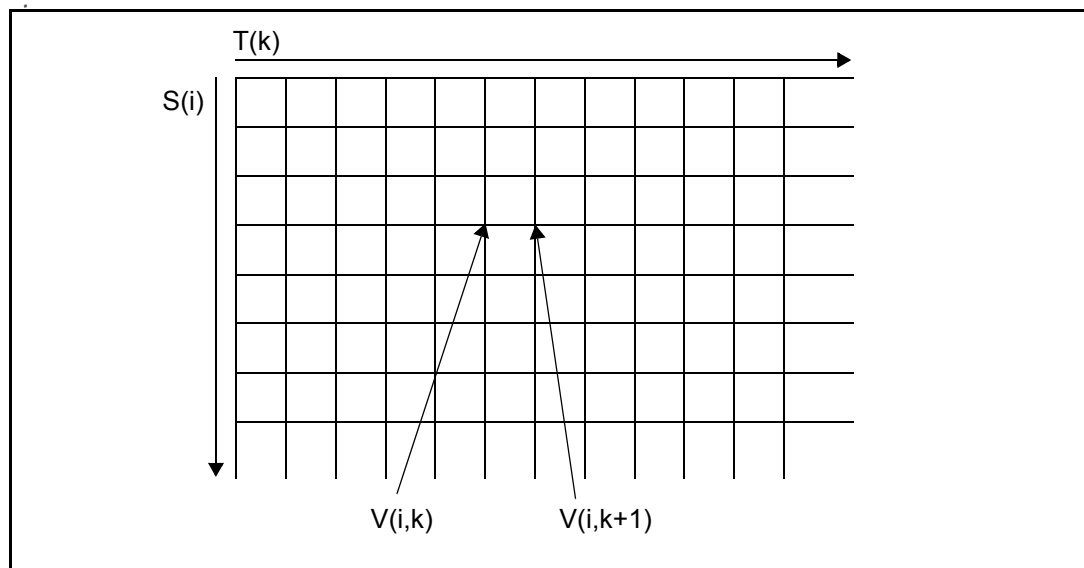


Figure 5.Asset and time discretization grid

The first order approximation to the time-derivative on the grid is

$$\frac{\partial V}{\partial t}(S, t) \approx \frac{V_i^k - V_i^{k+1}}{\delta t}$$

Similarly, the approximations for Delta (the first derivative with respect to S) and Gamma (the second derivative with respect to S) using central differences are as follows

$$\frac{\partial V}{\partial t}(S, t) \approx \frac{V_{i+1}^k - V_{i-1}^k}{2\delta t}$$

$$\frac{\partial^2 V}{\partial t^2}(S, t) \approx \frac{V_{i+1}^k - 2V_i^k + V_{i-1}^k}{\delta t^2}$$

The overall approximation for the Black-Scholes equations is:

$$1) \quad \frac{V_i^k - V_i^{k+1}}{\delta t} + \alpha_i^k \left(\frac{V_{i+1}^k - V_{i-1}^k}{2\delta t} \right) + \beta_i^k \left(\frac{V_{i+1}^k - 2V_i^k + V_{i-1}^k}{\delta t^2} \right) + \gamma_i^k V_i^k = O((\delta t, \delta S^2)^0)$$

The coefficients α and β are functions of the discretized asset value ($i\delta S$). This approximation can be rearranged to find the value of the option at the next time-step V_i^{k+1} from values of the option at the current time-steps V_i^k , V_{i-1}^k and V_{i+1}^k . This is the explicit finite differences method. Analysis of the convergence of parabolic PDEs (of which this is an example) under the explicit method shows that the time-step size cannot be chosen arbitrarily. Convergence depends on the values of coefficients α , β , γ and the size of asset grid step. The following inequality must be observed:

$$\delta t \leq \frac{1}{\sigma^2} \left(\frac{\delta S}{S} \right)^2$$

Typically the accuracy of the option price result is chosen by setting the size of the asset-price step. This places a bound on the size of time-step and the number of iterations necessary to price the option.

5.2 Initial code description

The size of the 1-d grid is set by NumSteps, and first two loops set up the grid of asset prices and the values in coefficients a , b , and c . Equation (1) has been re-arranged to group the coefficients of V_i^k , V_{i-1}^k and V_{i+1}^k . These coefficients are represented by the arrays a , b and c in the code.

The third loop sets up the boundary condition that corresponds to the application of a put at the expiry date.

$$\frac{\partial V}{\partial t}(S, T) = \max(0.0, X - S)$$

Finally, the main loop iterates backwards, from the solution at $t=T$ to $t = 0$. In this specific example, the loop could iterate forwards, as there are no time dependencies in coefficients. However, if we were to extend the model to include time variations in the coefficients a , b , c , such as using data from a yield curve, we must realize we are solving backwards in time and therefore iterate backwards. When pricing an American style option that allows exercise at any point in time, the put option must be exercised at each time-step.

The price of the option at $t=0$, is chosen at the centre of the grid, because that point corresponds to the original starting value of S .

5.3 Parallelizing sequential code

The simplest way to make use of the PE array is to "block" the 1-d grid over the PE array in as explained in the binomial tree example. `NumSteps` are chosen such that each PE holds a unique value in the grid. The time stepping kernel calculation requires option values from two adjacent points in the grid. These are supplied via swazzles up and down the array. The boundary conditions at each time-step are applied to the PEs at both ends of the PE array.

The code to apply the boundary conditions requires a `poly-if` statement. Instructions within a branch conditional on a poly value are predicated rather than conditionally executed. This means that the instructions are always executed on every PE, but on PEs where the poly condition is false, variable values are not updated. In the example below, the number of instructions executed on every PE is a sum of the instructions executed in all the branches, that is, $2 + 1 + 20 = 23$.

```
if( penum == 0 )
    {
        // 2 instructions
    }
else if( penum == __NUM_PES__ -1)
    {
        // 1 instruction
    }
else
    //if( (penum != 0) && (penum !=95) )
    {
        // 20 instructions
    }
}
```

5.4 Incorporating the vector math library

There is very little point in using vector instructions in this example in its current form. Using vector instructions normally requires unrolling loops to expose further parallelism. However, the process of converting code to run on the PE array has exposed all available parallelism.

5.5 Performance and possible improvements

The explicit method PDE solver is similar in some ways to trinomial tree methods (which are a direct extension of the binomial tree methods covered in the binomial tree example). The explicit method also suffers from similar problems. Once the inner loop (that updates the discretized grid) has been unrolled, there is no more parallelism that can be exploited as it is impossible to unroll the time-stepping loop. Parallelizing the problem across two CSX600s on an Advance card can also cause problems because of the data sharing needed at the boundary between data on each processor.

```
\>american_finitediff -v:reference
American Put (Explicit Finite Differences) value: 4.068666
American Put (Explicit Finite Differences) runtime: 0.046000secs
\>american_finitediff -v:poly
American Put (Explicit Finite Differences) value: 4.068666
American Put (Explicit Finite Differences) runtime: 0.079000secs
```

The problem for this particular grid size is that it runs at about 60% of the host speed. However, because only one of the CSX600 processors is utilized, it is possible to run a separate query on the other processor, therefore doubling the effective processing rate.

It is possible to increase the accuracy of the result by increasing the resolution of grid. We can block the data arrays across the PE array just as we did in the binomial example and this would show a performance improvement on a par with the binomial example.

For certain options (digitals, or barriers) increasing the step size can prevent oscillations, giving a stable result. However, the explicit method still places a bound on the time step size with respect to the asset step size. Doubling the asset step size means the time step size must quarter in size.

6 Broadie-Glasserman random tree method

6.1 Background

Pricing an American option in a Monte-Carlo framework is a substantially more complex problem compared with pricing path-dependent options ([Section 4: Asian option pricing via Monte Carlo on page 17](#)). The extra difficulty comes because when attempting to optimally solve the option, we must solve a problem known as an optimal stopping problem. This introduces not only conceptual difficulties, but requires significantly more computation. Most methods work with Bermudan options and assume that in the limit these will tend to an approximation of an American option.

The value of an American option at time $t=0$ over the lifetime $0 < t < T$, with the strike K , and S the underlying asset, is:

$$V(0) = \sup E[\max(e^{-rt}(K - S(t)), 0)]$$

Where \sup is the maximum value over the range of the function.

Given the expected value of the option at time t_{i+1} , and the value if the payoff is exercised at t_i , a decision can be made at each time-step to exercise if t_i is optimal. The value of the option when not exercising is called the continuation value.

There are various approaches to finding the expected future (time-discounted) value of the option given its current state. Longstaff and Schwartz use a series of fitted polynomial basis functions to give the predicted future value [\[12\]](#). We use Broadie and Glasserman's random tree approach. This scheme is quite complex so is only covered briefly in this document. Readers are referred to [\[13\]](#) for further information.

In any approach to estimating the continuation value, there are two sources of systematic error leading to bias. The first results from the use of future information in making the decision to exercise (which is inevitable in schemes that work backwards along simulated paths); this leads to high bias. The second approach however, provides a low bias. This results from a sub-optimal exercise decision.

The valuation of the option at the expiry date is insignificant, the item of interest ultimately is its value at $t = 0$. A dynamic programming formulation can be used to recursively determine the value at $t = 0$. The option $V_i(\chi)$ is the value of the option at t_i given the state $X_i = \chi$.

$$V_m = h_m(\chi)$$

$$V_{i-1}(\chi) = \max\{h_{i-1}(\chi), E[D_{i-1,i}(\chi_i)V_i(\chi_i) | X_{i-1} = \chi]\}$$

$$i = 1, \dots, m$$

$D_{i-1,i}$ is the discount factor from t_{i-1} to t_i and $h_m(\chi)$ is the payoff.

The random tree method involves simulating a tree of paths; at each node b independent successor states are generated, where b is the branching factor (≥ 2). The high and low estimators are defined by backward induction, starting at $t = t_m$ - the expiry date. At the terminal nodes

$$\tilde{V}_m = h_m(\tilde{X}_m)$$

Where the \sim notation implies payoff is applied over each of the terminals. The tree can be visualized as in [Figure 6](#).

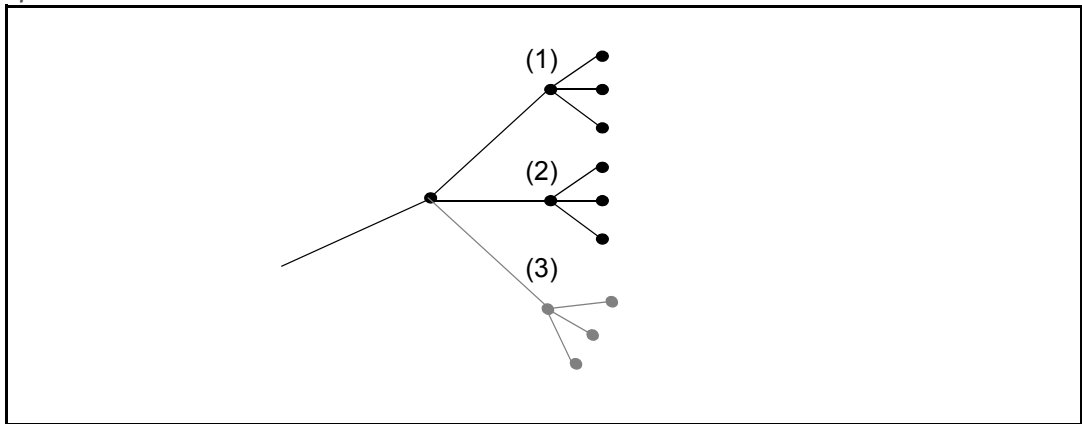


Figure 6. Multinomial tree with branching = 3

For the high estimator, apply backward induction and the nodes at level $m-1$ in the tree are valued as

$$\tilde{V}_{m-1} = \max \left\{ h_{m-1}(\tilde{X}_{m-1}), \frac{1}{b} \sum_{j=1}^b \tilde{V}_m \right\}$$

This can be intuitively understood as the arithmetic average of the future value over each successor branch. The low estimator is more complicated and calculating it is split into two parts. The continuation value is defined as

$$v = \alpha, \text{ if } Y_1 \leq \alpha \text{ else } Y_2$$

Where Y_1, Y_2 are the average of disjoint subsets of the successor nodes. In practice, Broadie and Glasserman use $(b-1)$ values to calculate Y_1 and use the remaining value as Y_2 . They then average the result over all b ways of leaving out one of the successor nodes.

Each estimator is calculated backwards recursively until the root node is reached. The steps described form the Monte-Carlo simulation kernel, so this kernel must be repeated a number of times and estimator values averaged.

6.2 Initial code description

The underlying asset dynamic is modeled as geometric Brownian motion (modeled by the `GBM(...)` function in the code following), so implicitly this is a lognormal process with $\mu = 0$ and $\sigma = 1$. The generation of random variates is controlled by the `NormInv()` function. Uniform variates are generated using calls to `drand48()` and transformed to normal

variates using the inverse cumulative normal distribution. The approximation used for inverse cumulative normal distribution is important, the implementation from Peter Acklam has full machine precision. Other approximations' precision may vary.

The storage requirements for a multinomial or "bushy" tree (with branching factor, 'B') grow exponentially with the number of exercise days, 'D'. This can quickly lead to excessive memory use. However, as Broadie and Glasserman themselves point out, it is not necessary to store the entire tree. The processing of the tree implicitly occurs in depth-first order (processing the leaves first), so the maximum memory requirements can be reduced to B*D elements, rather than BD. The current path in the tree is held in the array v. The branch being processed at tree depth, d, is indexed using the array w. The value at a leaf is generated with the code;

```

// exercise the option
y = z * (v[w[d]][d] - X);
v[w[d]][d] = (y > 0.0) ? y : 0.0;
// whilst not generated all leaves
if (w[d] < B)
{
    v[w[d] + 1][d] = v[w[d - 1]][d - 1] *
exp(GBM(drift, SigSqrt));
    w[d] = w[d] + 1;
}

```

The tree held in v can be visualized as follows;

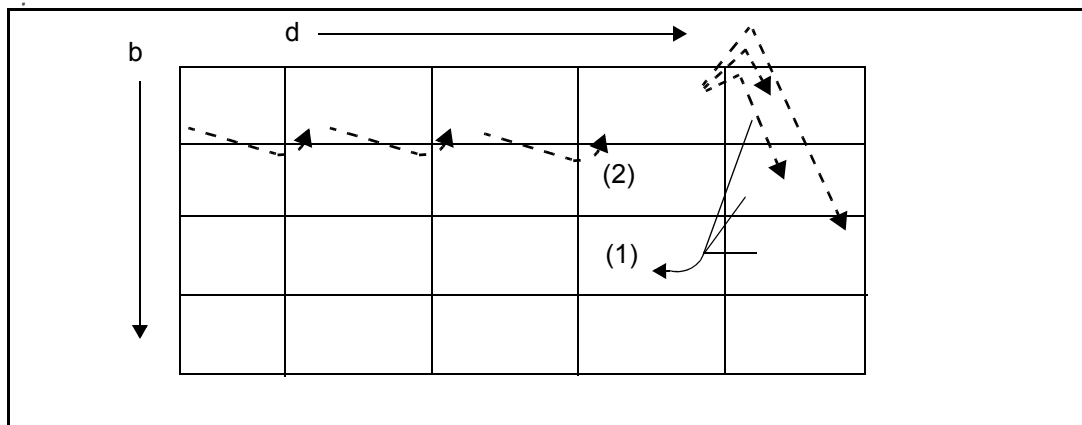


Figure 7. Multinomial tree with branching = 3

The dashed lines show the evolution of the price path before the high estimator is calculated for node (1). Calculating the estimators for node (2) requires re-evolving the price path onto the leaves and placing the results in the element adjacent to (1).

6.3 Parallelizing sequential code

Now use a technique similar to the previous Monte-Carlo example. This includes converting all functions to act on poly variables (or to be a poly function) and performing a reduction in one form or another to get the data back into mono memory.

Each PE now performs a fraction of the number of simulations. The number of simulations has been chosen such that it is not a multiple of 96. Unfortunately, this now means that more than nSimulations will be performed. In order to guard against this, restrict the execution of the simulation kernel on some of the PEs. This is a common technique.

```

current_sim = get_penum();
    for (Simulation = 0; Simulation < nSimulations;
Simulation+=__NUM_PES__)
    {
// rest of code..

// ensure that only nSimulations are performed
if( current_sim < nSimulations )
    {
        EstimatorSump = EstimatorSump + v;
    }
    current_sim += __NUM_PES__;
    }

```

6.4 Incorporating the vector math library

As with most Monte-Carlo schemes, the code can take advantage of massive data-parallelism available by using the vector functions supplied in the Vector Math Library. Follow the same conversion process outlined in the Asian Monte-Carlo example:

1. Unroll the simulation loop a further four times.
2. Convert poly variables to `__DVECTORS` and substitute the appropriate intrinsics for operators.

6.5 Performance and possible improvements

Performance

As in the previous Monte-Carlo method example, the performance increase depends upon the number of simulations. A small number of simulations numbering just thousands is the cut-off, below which no acceleration is seen. As the number of simulations is increased, to increase the accuracy of the result, the observed speedup increases.

```

\>broadieglasserman -v:reference
American Call (Broadie-Glasserman Tree Method) value: 7.732528
American Call (Broadie-Glasserman Tree Method) runtime:
3.844000secs
\>broadieglasserman -v:vector
American Call (Broadie-Glasserman Tree Method) value: 7.734168
American Call (Broadie-Glasserman Tree Method) runtime:
0.875000secs

```

As you can see, for 38400 trials (the default, if not specified on the command-line) the speed-up is x4.4. Increasing the number of trials to 3,840,000 shows a x6 speed-up. This increase in speedup is due to the decreasing proportion of time spent in transferring data to and from the Advance board.

```
\>broadieglasserman -v:reference -s:3840000
American Call (Broadie-Glasserman Tree Method) value: 7.729304
American Call (Broadie-Glasserman Tree Method) runtime:
38.501000secs
\>broadieglasserman -v:vector -s:3840000
American Call (Broadie-Glasserman Tree Method) value: 7.730991
American Call (Broadie-Glasserman Tree Method) runtime:
6.437000secs
```

7 Summary

These examples should provide confidence in approaching other financial algorithms with a view to exploiting the CSX600 processor. It is important to find parallelism strategies that minimize communication between processing elements and minimize the movement of data between the mono memory and poly memory.

Monte Carlo pricing approaches almost always parallelize because each simulation is independent of the others. Lattice methods, for example binomial trees and PDEs, can benefit from SIMD parallelism, but the grid sizes must be chosen judiciously. If spreading the algorithm across two CSX600 processors requires excessive communication, two independent problems can be run, one on each processor.

Lastly, a high capacity approach can be used for computationally intensive analytic solutions to the Black-Scholes differential equation. Although running a single Black-Scholes analytic solution on the Advance card may be no faster than running it on an Intel Xeon (or AMD Opteron), when attempting to run 1,000,000 solutions back-to-back the Advance card can be significantly faster.

Once the correct coarse-grain parallelism strategy has been found, ensure the optimized math libraries are used. Try to find even more parallelism by unrolling loops and using `__DVECTOR` data-types that perform four calculations at a time. Finally, investigate the use of the ClearSpeed Visual Profiler to profile your code.

8 Bibliography

1. ClearSpeed Visual Profiler User Guide
2. ClearSpeed SDK Reference Manual
3. The Cn Standard Library
4. CSX600 Runtime Software User Guide
5. Options, Futures and Other Derivatives (Fourth Ed.) - John C. Hull (Pub Prentice Hall 2000)
6. Abramowitz and Stegun: A Handbook of Mathematical Functions (<http://www.math.sfu.ca/~cbm/aands/>)
7. Cox, J., Ross, S., & Rubinstein M., - "Option Pricing: A Simplified Approach." Journal of Financial Economics, 7. (Sept '79)
8. Monte Carlo Methods in Finance - Peter Jaeckel (Pub. Wiley Finance Reprint 2003)
9. Options: Approach for Parallel Implementation of Boyle's Monte Carlo Method - R. Mirani (<http://www.datasimfinancial.com/articles.php>)
10. Paul Wilmott on Quantitative Finance 2nd Ed - Paul Wilmott (Pub. Wiley Finance Reprint 2003)
11. Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach - Daniel J. Duffy (Pub Wiley Finance 2006)
12. Valuing American Options by Simulation: A Simple Least-Squares Approach - Francis A. Longstaff & Eduardo S. Schwartz (<http://galton.uchicago.edu/~mykland/346W05/Longstaff.pdf>)
13. Monte Carlo Methods in Financial Engineering v.53 - Paul Glasserman (Pub. Springer 2000)

Appendix A Reduction methods

Reduction operations are common in multi-processing environments. It is often necessary to find the sum or product (or some other simple 2-to-1 function) of a list of values distributed over the processors. For instance, at the end of a calculation, we may have 96 results, one on each PE. In order to sum these partial results to a single value, a sum-reduce must be performed.

Note: There are no reduction functions in the SDK 2.50 release (future releases will contain them).

There are two main reduction methods for the CSX600 architecture, and they vary in their efficiency.

The first relies on copying the values from poly memory into mono memory and performing the sum in mono memory. The pseudo-code follows:

```
poly double partial_result;
double p_result[96];
double result = 0.;
// ... perform calculation and write to partial_result
memcpy2m( p_result, &partial_result, sizeof(double);
for(i=0;i<96;i++) result += p_result[i];
    // result contains the sum-reduce value
```

Even allowing for slight inefficiencies in the code this method takes thousands of clock cycles to complete. This is an inefficient method because only the mono arithmetic unit is used and this disregards the processing power of the PE array.

The second method presented is more efficient and with sensible optimization can be made extremely fast. This method relies on the swizzle path connecting adjacent PEs. For more details on swizzling, see [\[1\]](#) and [\[2\]](#).

```
poly double partial_result;
double result = 0.;
// ... perform calculation and write to partial_result
for (i = 0; i < __NUM_PES__; i++)
{
    partial_result = swizzle_down_double(partial_result);
    result += get_swizzle_low_double();
}
```

The `swizzle_down_*` function takes value in the register file on PE(n) and writes it into the register file on PE(n-1). Zeros are shifted into the register on PE95. As values are shifted out PE0, they can be picked up using `get_swizzle_low_*`. The swizzle path is extremely fast, taking only 2 clock cycles to transfer a double word between PEs.

Appendix B `cs_vgaussian` stack overflow

The implementation of `cs_vgaussian` in SDK 2.50 contains a defect that sometimes causes a stack overflow in the processor. As a workaround, the following code can be used in its place.

```
void gaussrandv(cs_rand48_stream * rngstream, __DVECTOR * x,
__DVECTOR * y )
{
    poly double two_pi = 2.*M_PI;

    x1 = cs_vdrand48(rngstream);
    x2 = cs_vdrand48(rngstream);
    w = cs_sqrtp( -2.0 * cs_logp( x1 ) );
    *x = __cs_vmul( w , cs_cosp( __cs_vmul_scalar(x2,two_pi) ) );
    *y = __cs_vmul( w , cs_cosp( __cs_vmul_scalar(x2,two_pi) ) );
}
```

Revision history

Date	Revision	Changes
July 2007	1.0	Details of revision status.

Table 1. Document revision history

ClearSpeed Technology, Inc.
3031 Tisch Way, Suite 200
San Jose, CA 95128
United States of America

Tel: +1 408 557 2067
Fax: +1 408 557 9054

Email: info@clearspeed.com

Web: <http://www.clearspeed.com>

Support: <http://support.clearspeed.com>

ClearSpeed Technology plc
3110 Great Western Court
Hunts Ground Road
Bristol BS34 8HP
United Kingdom

Tel: +44 (0)117 317 2000
Fax: +44 (0)117 317 2002

1. Information and data contained in this document, together with the information contained in any and all associated ClearSpeed documents including without limitation, data sheets, application notes and the like ('Information') is provided in connection with ClearSpeed products and is provided for information only. Quoted figures in the Information, which may be performance, size, cost, power and the like are estimates based upon analysis and simulations of current designs and are liable to change.
2. Such Information does not constitute an offer of, or an invitation by or on behalf of ClearSpeed, or any ClearSpeed affiliate to supply any product or provide any service to any party having access to this Information. Except as provided in ClearSpeed Terms and Conditions of Sale for ClearSpeed products, ClearSpeed assumes no liability whatsoever.
3. ClearSpeed products are not intended for use, whether directly or indirectly, in any medical, life saving and/ or life sustaining systems or applications.
4. The worldwide intellectual property rights in the Information and data contained therein is owned by ClearSpeed. No license whether express or implied either by estoppel or otherwise to any intellectual property rights is granted by this document or otherwise. You may not download, copy, adapt or distribute this Information except with the consent in writing of ClearSpeed.
5. The system vendor remains solely responsible for any and all design, functionality and terms of sale of any product which incorporates a ClearSpeed product including without limitation, product liability, intellectual property infringement, warranty including conformance to specification and or performance.
6. Any condition, warranty or other term which might but for this paragraph have effect between ClearSpeed and you or which would otherwise be implied into or incorporated into the Information (including without limitation, the implied terms of satisfactory quality, merchantability or fitness for purpose), whether by statute, common law or otherwise are hereby excluded.
7. ClearSpeed reserves the right to make changes to the Information or the data contained therein at any time without notice.

© Copyright ClearSpeed Technology plc 2007. All rights reserved.

Advance is a registered trademark of ClearSpeed Technology plc

ClearSpeed, ClearConnect, Advance and the ClearSpeed logo are trade marks or registered trade marks of ClearSpeed Technology plc. All other brands and names are the property of their respective owners.