BOPM implemented on a GPU-architecture

Kristoffer Fürst
Abstract

We used the Binomial Options Pricing Model (BOPM) implemented on a Graphics Processing Unit (GPU) to calculate the value of European and American options, of both put and call type. The advantage of using a GPU over a CPU is that a GPU has many more processing-cores than a CPU and can perform more calculations simultaneously. This, of course, also implies limitations.

The result was compared with both the exact value and with the value obtained using a Central Processing Unit (CPU). The CPU-code was implemented using C++ and the GPU-code was implemented using NVIDIA’s programming language CUDA. It turns out that the value obtained using the GPU is for the most part almost consistent with both the exact value and the value obtained using a CPU.
Contents

1 Introduction 1

2 Basic theory of vanilla options 1

3 BOPM - Binomial Option Pricing Model 2
   3.1 $u = \frac{1}{2}$ 2
   3.2 $p = \frac{1}{2}$ 3
   3.3 Algorithm 3
      3.3.1 steps 1 3
      3.3.2 steps 2 4
      3.3.3 steps 3 4

4 CPU-implementation 4
   4.1 European put 4
      4.1.1 CPU European put, Test1 5
      4.1.2 CPU European put, Test2 5
      4.1.3 CPU European put, Test3 5
      4.1.4 CPU European put, Test4 5
      4.1.5 CPU European put, Test5 5
      4.1.6 CPU European put, Test6 6
      4.1.7 Conclusion 6
   4.2 American put 6
      4.2.1 CPU American put, Test1 6
      4.2.2 CPU American put, Test2 7
      4.2.3 CPU American put, Test3 7
      4.2.4 CPU American put, Test4 7
      4.2.5 Conclusion 7
   4.3 American call 7

5 The theory behind GPUs and CUDA 7

6 GPU-implementation explained 8
   6.1 European options 8
   6.2 American options 10

7 test cases for GPU-implementation 10
   7.1 European put 11
      7.1.1 GPU European put, Test1 11
      7.1.2 GPU European put, Test2 11
      7.1.3 GPU European put, Test3 11
      7.1.4 GPU European put, Test4 11
      7.1.5 GPU European put, Test5 12
      7.1.6 GPU European put, Test6 12
      7.1.7 Conclusion 12
   7.2 American put 12
      7.2.1 GPU American put, Test1 13
      7.2.2 GPU American put, Test2 13
      7.2.3 GPU American put, Test3 13
      7.2.4 GPU American put, Test4 13
      7.2.5 Conclusion 13
   7.3 American call 13

8 Conclusion 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C++ code for European put</td>
<td>14</td>
</tr>
<tr>
<td>B</td>
<td>C++ code for American option</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>CUDA code for European put</td>
<td>16</td>
</tr>
<tr>
<td>D</td>
<td>CUDA code for American put</td>
<td>18</td>
</tr>
</tbody>
</table>
1 Introduction

In this thesis, I will have a closer look at the binomial option pricing model (BOPM) and how to implement it using a graphics processing unit (GPU). The options I have chosen to look at in detail are the American put and American call. First, the theoretical background will be covered in short, which includes the basics of vanilla options and the assumptions that are made. To make it easier for the reader to follow the discussion, I will begin with European options and then expand to include American options.

Thereafter, the BOPM is discussed in greater detail and an algorithm is developed. This part will not discuss any implementation details, and is thus independent of programming language preferences.

The implementation part follows next, which starts out with a discussion of how to implement the BOPM on a central processing unit (CPU). I have chosen to include this intermediary to make the implementation on a GPU easier to understand. The programming language I have chosen to use is C++ for reasons explained in section 5. Finally, we arrive at the reason for using a GPU and how the actual implementation on such a device is done.

The reader is assumed to have knowledge of the fundamental mathematics on University level and at least some basic understanding of iterative programming. I discuss the theory only superficially, for a deeper understanding I refer the reader to [3].

From NVIDIA’s CUDA-site, [1], I have downloaded [2]. In that document, Victor Podlozhnyuk has implemented a European option with CUDA-code. CUDA is a programming language developed by NVIDIA designed to make use of the computational powers of GPUs and is the language I will use for the final implementation.

2 Basic theory of vanilla options

The simplest form of a financial derivative is the European call option, which gives its holder the right to buy an underlying asset for a preassigned price (strike price) at a specified time in the future. The writer is, if the holder chooses to exercise the option, obligated to deliver the underlying asset. A European put is almost the opposite, it gives its holder the right to sell an underlying asset for a preassigned price at a specified time in the future. In the latter case, the writer is, if the holder chooses to exercise the option, obligated to buy the underlying asset paying the strike price.

An American option is also considered a vanilla option (the word vanilla means that the option is of a simple nature). The only difference from the European options discussed above is that there is no specified exercise date. With American options, there is just a "use before" date. This means that the holder has the right to exercise the option any time before this date, but not after. Of course, since this gives the holder a more flexible product, the price is often higher than that for an European option.

Under the assumption that the history gives no more information than the current value of the underlying asset and that new information about the underlying asset immediately affects its value, we can form a continuous random walk

$$\frac{dS}{S} = \sigma dX + r dt$$  \hspace{1cm} (2.1)

Where $S$ is the value of the underlying asset at time $t$, $dX$ is a sample from the normal distribution, $\sigma$ is the volatility and $r$ is the growth rate. This continuous random walk will
later be discretized, and used in the algorithm for the BOPM. Furthermore, we assume the existence of a risk-free investment. This could, for example, be a government bond (in reality, not even a government bond is totally risk-free).

This is in fact all the theory we need to move on to the binomial option pricing model.

3 BOPM - Binomial Option Pricing Model

If we know the value of the asset, denoted by \( S \) from now on, today (which we usually do and are assumed to do from now on), we divide the life-span of the option (T) into small pieces \( dt = T/NUM\_STEPS \) where \( NUM\_STEPS \) is the number of time-steps taken. It is clear that we have to choose how many parts we want to divide the life-span into (if we would choose \( \infty \), we would not need a discrete walk). We want to choose a time-step as large as possible, but the computation time complexity of this algorithm grows like \( O(NUM\_STEPS^2) \), so we will have to make a compromise between the number of time-steps and the accuracy of our computation.

The probability of the price moving up \( (p) \) or down \( (1-p) \) has to be known and also, the price can only move to two preassigned prices in each step. If the price moves up, it moves to \( uS \) (which is \( > S \) ) and if it moves down, it moves to \( dS \) (which is \( < S \) ). \( u \) and \( d \) are coefficient where, of course, \( u > 1 \) and \( d < 1 \). We choose these variables so that the statistical properties, i.e. the mean and variance, from (2.1) are kept.

This will create a tree of asset values, and since a move up followed by a move down is the same as the other way around, there will for \( i \) steps in time be only \( i+1 \) possible asset values. So the value of the option at time \( i \) can be written

\[
V_i = E[e^{-r dt}V_{i+1}]
\]

Which is actually just another way of interpreting Black-Scholes formula (not discussed in this article). Simple algebra can be used to show that

\[
pu + (1-p)d = e^{r dt}
\]
\[
pu^2 + (1-p)d^2 = e^{(2r+\sigma^2)dt}
\]

A third equation is needed to determine \( p, d \) and \( u \). Here we can make a choice, and among the most common (according to [3]) are \( u = 1/d \) and \( p = 0.5 \). Personally, I prefer the latter because I find it more intuitive, but we will use both in different implementations later to show how this is done (the difference between them is in practice negligible for small stepsizes). In the two subsections below we present the equations for computing \( p, d \) and \( u \).

3.1 \( u = \frac{1}{d} \)

\( A \) is an intermediary to make the formulas easier to comprehend.

\[
A = \frac{1}{2}(e^{-r dt} + e^{(r+\sigma^2)dt})
\]
\[
d = A - \sqrt{A^2 - 1}
\]
\[
u = A + \sqrt{A^2 - 1}
\]
\[
p = \frac{e^{r dt} - d}{u - d}
\]
3.2 \quad p = \frac{1}{2}

\begin{align*}
d &= e^{rt}(1 - \sqrt{e^{2}\sigma^2 dt} - 1)\\
u &= e^{rt}(1 + \sqrt{e^{2}\sigma^2 dt} - 1)\\
p &= \frac{1}{2}
\end{align*}

Since \( S \) is the value of the underlying asset, we denote \( S^i_j \) as the value of the asset at time-step \( i \) where \( j \) is an integer (0 \( \leq \) \( j \) \( \leq \) \( i \)) used to denote the \((i + 1)\) possible values at time-step \( i \). Thus,

\[ S^i_j = d^{i-j} u^j S^0_0 \]  

(3.1)

where \( S^0_0 \) is the value of the asset today.

Before we can move on to the actual algorithm, the way to value the option must be determined. This is very simple in the cases European of puts and call. For both puts and calls, we get

\[ V^i_j = e^{-r dt}(p V^{i+1}_{j+1} + (1 - p) V^{i+1}_{j}) \]  

(3.2)

It gets a little more tricky when we come to American options, since if there would be a greater profit to be made from exercising the option and putting the money in the bank, we would of course do this. This means that the value of an American put is

\[ V^i_j = \max(\max(E - S^i_j, 0), e^{-r dt}(p V^{i+1}_{j+1} + (1 - p) V^{i+1}_{j})) \]  

(3.3)

and the value of an American call is

\[ V^i_j = \max(\max(S^i_j - E, 0), e^{-r dt}(p V^{i+1}_{j+1} + (1 - p) V^{i+1}_{j})) \]  

(3.4)

3.3 Algorithm

1. Build a tree of possible values of the asset.
2. Calculate the values of the option at expiry.
3. Go back up the tree and evaluate the option at all times before expiry.

This is the general algorithm for the BOPM and it can be used for both European and American options. The complexity in time will be \( O(NUM\_STEPS^2) \) as mentioned before. Because we have an explicit formula for calculation of the asset price we actually do not need to store and as far as the value of the option goes, we are only interested in the value of the option today. This means we can forget the values as we work our way back up the tree. Hence, the complexity with respect to data is \( O(NUM\_STEPS) \)

3.3.1 steps 1

We actually just need the values of the asset at expiry at this point, since we do have an explicit formula for the value of the asset and can calculate it no matter what node in the tree we’re at. This is simply given by 3.1 where \( i \) is replaced by \( NUM\_STEPS \).
3.3.2 steps 2

For a put (European or American)

\[ V_i^j = \max(E - S_i^j, 0) \]

For a call (European or American)

\[ V_i^j = \max(S_i^j - E, 0) \]

3.3.3 steps 3

We have to go through every possible value of the option for every time-step. This means that at exercise, we have \( \text{NUM\_STEPS} + 1 \) calculations to do. This amount decreases by 1 for every time-step we go back. Each calculation is given by 3.2, 3.3 or 3.4.

4 CPU-implementation

The implementation of BOPM is pretty straightforward, at least when using a CPU. The steps from the algorithm presented in section 3.3 are implemented in order. The first two steps are done in one common for-loop and causes no difficulties. The last step on the other hand is a bit more tricky, in the sense that one needs to keep track of what for-loop (this step needs two for-loops) represents what. The outermost loop is the loop stepping back in time, and the innermost loop steps through every possible value at the time determined by the outermost loop.

4.1 European put

First, we implemented the algorithm for a European option (put), since this is simpler than that for an American option. The code is found in appendix A on page 14. To run the program, you call

\[ > \text{euro.exe } S_0 \ T \ r \ \text{vol} \ \text{NUM\_STEPS} \]

where \( S_0 \) is the value of the underlying asset today, \( T \) is the time until expiry, \( r \) is the interest rate, \( \text{vol} \) is the volatility and \( \text{NUM\_STEPS} \) is the number of steps to be taken (explained above).

Following are a couple of tests to make sure my program functions correctly. I have chosen to use the same input data as is used in figure 10.5 in [3], so that I can compare the results. This means using \( u = 1/d \) rather than \( p = 1/2 \).

<table>
<thead>
<tr>
<th>Interest rate (r)</th>
<th>0.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strike price (E)</td>
<td>10</td>
</tr>
<tr>
<td>Asset value (S)</td>
<td>5</td>
</tr>
<tr>
<td>Volatility (( \sigma ))</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 1: These values will be used in the first 4 tests for the European put in this section.
### 4.1.1 CPU European put, Test1

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (NUM_STEP)</td>
<td>16</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.5 in [3]</td>
<td>4.4292</td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>4.42925</td>
</tr>
</tbody>
</table>

### 4.1.2 CPU European put, Test2

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>32</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.5 in [3]</td>
<td>4.7047</td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>4.7047</td>
</tr>
</tbody>
</table>

### 4.1.3 CPU European put, Test3

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>128</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.5 in [3]</td>
<td>4.8511</td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>4.85113</td>
</tr>
</tbody>
</table>

### 4.1.4 CPU European put, Test4

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>256</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.5 in [3]</td>
<td>4.5636</td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>4.56349</td>
</tr>
</tbody>
</table>

### 4.1.5 CPU European put, Test5

Here, I compare to the exact value, and change a couple of parameters.

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interest rate (r)</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Strike price (E)</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Asset value (S)</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Volatility (σ)</td>
<td>0.4</td>
</tr>
<tr>
<td>Output</td>
<td>Exact value</td>
<td>13.15</td>
</tr>
<tr>
<td></td>
<td>from my program (1024 steps)</td>
<td>13.1362</td>
</tr>
<tr>
<td></td>
<td>from my program (4096 steps)</td>
<td>13.1591</td>
</tr>
</tbody>
</table>
4.1.6 CPU European put, Test6

This test, as the previous, also compares the program’s result to a exact result, but with different parameters.

<table>
<thead>
<tr>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>years until expiry (T)</td>
<td>0.41096</td>
</tr>
<tr>
<td>Interest rate (r)</td>
<td>0.04</td>
</tr>
<tr>
<td>Strike price (E)</td>
<td>340</td>
</tr>
<tr>
<td>Asset value (S)</td>
<td>350</td>
</tr>
<tr>
<td>Volatility (σ)</td>
<td>0.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact value</td>
<td>10.63</td>
</tr>
<tr>
<td>from my program (512 steps)</td>
<td>10.8364</td>
</tr>
<tr>
<td>from my program (6144 steps)</td>
<td>10.6808</td>
</tr>
</tbody>
</table>

4.1.7 Conclusion

Based on the premise that the values in [3] are correct, my program seems to be functioning correctly (test1-test4) except for a small error in the first and last test. This is most possibly due to the fact that arithmetic on a computer is not associative.

Looking at test5 and test6, we can see that the computer produces a value sufficiently close to the exact value.

4.2 American put

The code for the CPU-implementation used to calculate the value of an American put is found in appendix B on page 15. In this case, I have used \( p = \frac{1}{2} \). To run the program, you call

\[ > \text{amer.exe } S_0 \ T \ r \ vol \ NUM\_STEPS \ type \]

where \( S_0 \) is the value of the underlying asset today, \( T \) is the time until expiry, \( r \) is the interest rate, \( vol \) is the volatility and \( NUM\_STEPS \) is the number of steps to be taken (explained above). \( type \) is either ‘put’ or ‘call’ depending on what kind of option you want to evaluate.

<table>
<thead>
<tr>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interest rate (r)</td>
<td>0.06</td>
</tr>
<tr>
<td>Strike price (E)</td>
<td>10</td>
</tr>
<tr>
<td>Asset value (S)</td>
<td>9</td>
</tr>
<tr>
<td>Volatility (σ)</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 2: These values will be used in all the tests for the American put.

4.2.1 CPU American put, Test1

<table>
<thead>
<tr>
<th>Input</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>years until expiry (T)</td>
<td>1</td>
</tr>
<tr>
<td>steps in time (steps)</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>from figure 10.9 in [3]</td>
<td>1.4473</td>
</tr>
<tr>
<td>from my program</td>
<td>1.44726</td>
</tr>
</tbody>
</table>
4.2.2 CPU American put, Test2

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>32</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.9 in [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>1.2590</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.259</td>
</tr>
</tbody>
</table>

4.2.3 CPU American put, Test3

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>128</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.9 in [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>1.1258</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.12586</td>
</tr>
</tbody>
</table>

4.2.4 CPU American put, Test4

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>steps in time (steps)</td>
<td>256</td>
</tr>
<tr>
<td>Output</td>
<td>from figure 10.9 in [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>from my program</td>
<td>1.3541</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.35407</td>
</tr>
</tbody>
</table>

4.2.5 Conclusion

As in the case with the European put, I get a small error, but in this case just in one of the cases, namely the third. Again, this is most likely due to the fact that arithmetic on a computer is not associative.
The American put can not be evaluated exactly, and hence we can not compare the value obtained with our program to an exact value.

4.3 American call

To find the value of an American call, the only change needed is to change $E - S$ to $S - E$ in the payoff-function. Thus, I believe test-cases are a bit of a push and have omitted such. And also, the American call will have the same value as an European call because in theory, it will never be exercised before expiry. In my implementation found in appendix B, I have included the possibility to choose either a put or a call.

5 The theory behind GPUs and CUDA

Since a GPU is (obviously) designed for computations associated with graphical visualization, there will be special restrictions on the computations. Despite this, there are advantages of using a GPU for some types of calculations. Since the GPU consists of many parallel
processors, it can operate on many threads at the same time and hence do many calculations simultaneously. This leads to the possibility of significant speedups in execution time.

The GPU can be interpreted as a big room of mathematicians each doing the same calculations but on different data. The instructions they follow are called kernels and they are performed on different data (or elements). Because of this independency, we can’t have static data. Also, only one kernel can run at a time on one GPU. Every process consist of reading data, operating on it, and then writing results. Hence, an optimal application for GPU-implementation have large sets of data, a high level of parallelism and minimal dependence between data elements.

As an example, one can think of the body of a for-loop with fewer loops than the maximum number of threads available on the GPU. When implemented on the GPU, every lap of the for loop becomes a separate thread. One can of course still use a for-loop over the thread’s individual IDs and thus control each thread individually and still take advantage of the power of parallelism.

The programmer needs a way of communicating with the GPU. There are several ways of doing this and I will use CUDA. CUDA (Compute Unified Device Architecture) is a programming language developed by NVIDIA, designed for implementation on GPUs. There exists several versions of CUDA, and the one I will use is the most common and consists of just a small set of extensions to the programming language C. This is the reason for using C++ in my CPU-implementation; the basis of the code can be reused. For more specifics of the specifics of CUDA, see [1].

6 GPU-implementation explained

Here the details of the CUDA code, found in appendix C and D, will be explained. We will begin with the code for a European option and then consider the American option. Fragments of code will follow every piece of explanation. To get an overview of the code the reader might want to have a look at the complete code in appendix C on page 16.

6.1 European options

First of all, memory to hold the value of the option to day is allocated (this is all we will need to save from the computation). This call is similar to C’s malloc and is called cudaMalloc. The command cudaThreadSynchronize() makes sure that every thread has completed its previous tasks. CUDA_SAFE_CALL is used when debugging, it checks for errors. The program would run just fine without the aforementioned command.

```c
CUDA_SAFE_CALL( cudaMalloc((void **)&d_value, sizeof(float)) );
CUDA_SAFE_CALL( cudaThreadSynchronize() );
```

Next, the number of blocks are set. A block can be thought of as a container. Every block contains a number of threads which all have access to a shared memory within the block. The number of threads per block (called "grid") is limited, which is why we might need to divide the threads into more than one block. "dim3" is CUDA’s variable type used to specify the number of blocks and grids. The function is executed as a usual C-function but with the number of blocks and threads per blocks specified.

```c
dim3 block(cell(NUM_STEPS/GRID_N),1);
dim3 grid(GRID_N,1);
euro_option<<<block, grid>>>(S0, E, u, d, r, dt, d_value, vol);
```
We now follow the execution into the function `euro_option`. Since the size of shared memory is limited, we have to do the calculations in steps. A part of the tree is loaded into a shared memory buffer. The code-segment below shows how the shared memory is initialized. Why two buffers are used will be explained later.

```c
__shared__ float dataA[CACHE_SIZE];
__shared__ float dataB[CACHE_SIZE];
```

The next piece of code we look at is the first two steps of the algorithm in section 3.3. Here, the code is consistent with that for the CPU-implementation except for the for-loop specifications. `threadIdx.x` gives the ID of the thread within a block (from 0 to `grid - 1`). Since `NUM_STEPS` might be bigger than `grid - 1`, we need some way of calculating values in the interval `[grid - 1 : NUM_STEPS]`. This is done by stepping one block ahead and checking if we have gone too far or not. For example, suppose that `threadIdx.x` is 4, `blockDim.x` (which is the same as the number grid above) is 10 and `NUM_STEPS` is 25. Then we should do the calculations inside the loop 25 times total but we only have 10 threads. The trick is then to let thread no.4 perform calculations for `i = 4`, `i = 14` and `i = 24`.

```c
for(int i = threadIdx.x; i <= NUM_STEPS ; i += blockDim.x){
    float price = S0*pow(u,i)*pow(d,NUM_STEPS-i);
    results[i] = fmax(E-price, 0);
}
```

Since we have divided our global data into smaller pieces of shared data, we can only do a limited number of computations at a time. This is handled by dividing the double-loop in step 3 of the algorithm in section 3.3 into two double-loops. The content of the outermost double-loop (found in the code-segment below) is run using all the shared memory. First, the elements that are to be operated on are read from global memory into the much faster shared memory. Next, the computations inside the loop 25 times total but we only have 10 threads. The trick is then to let thread no.4 perform calculations for `i = 4`, `i = 14` and `i = 24`.

```c
for(int i = NUM_STEPS; i >= 0 ; i -= CACHE_DELTA)
    for(int c_base = 0 ; c_base <= i + 1; c_base += CACHE_STEP){
        int c_start = min(CACHE_SIZE - 1, i - c_base);
        int c_end = c_start - CACHE_DELTA;
        __synchronize();
        for(int k = threadIdx.x; k <= c_start ; k += blockDim.x)
            dataA[k] = results[c_base + k];
        // contents
        __synchronize();
        for(int k = threadIdx.x; k <= c_end ; k += blockDim.x)
            results[c_base + k] = dataA[k];
    }
```

Next, we do calculations on the selected part of the double-loop. The loops walks through the time-steps and values at their disposal, and does the same calculations as in the CPU-implementation. What is most noteworthy here is the double buffering. We have to double buffer here since one thread might do its calculations and write the result, and then another thread reads the written value instead of the old value. And thus we would get the correct calculation carried out on the wrong data. This can be solved by reading from one buffer and writing to another, and forcing threads to wait for each other between every calculation thus forcing them to be at most one calculation apart.

```c
for(int k = c_start - 1; k >= c_end;)
    __synchronize();
    for(int l = threadIdx.x; l <= k ; l += blockDim.x)
        // contents
```
} k--;
__synchronize();
for(int l = threadIdx.x; l <= k; l += blockDim.x)
} k--;
}

The following code writes today’s value to the destination buffer. This needs to be done because the array results can not be copied to main memory. Since we are only interested of the value of the option today, this is all we need to save from the computations.

if(threadIdx.x == 0) *d_value = results[0];

When the function euro_option is done, we check that everything has gone according to plan (the first line below). Then we copy the value of the option to main memory and use cudaFree to free the memory that we allocated before we ran euro_option.

CUT_CHECK_ERROR("euro_option() execution failed\n");
CUDA_SAFE_CALL(cudaMemcpy(h_value, d_value, sizeof(float),
cudaMemcpyDeviceToHost));
CUDA_SAFE_CALL(cudaFree(d_value));

6.2 American options

In the case of an American option we have to do the same consideration as before. It might be better to exercise the option and put the money in the bank. The only place in the code that the American option differs from the European option the one found below. There are no surprises here, as this difference is the same as for the CPU-implementation.

for(int k = c_start - 1; k >= c_end;)
    __synchronize();
for(int l = threadIdx.x; l <= k; l += blockDim.x)
    float hold = expf(-r*dt)*(p*dataA[l+1]+(1-p)*dataA[l]);
    float price = S0*pow(u,c_base+l)*pow(d,1-(c_start-k)-c_base-l);
    float payoff = fmaxf(E-price, 0);
    dataB[l] = fmaxf(hold, payoff);
} k--;
__synchronize();
for(int l = threadIdx.x; l <= k; l += blockDim.x)
    float hold = expf(-r*dt)*(p*dataB[l+1]+(1-p)*dataB[l]);
    float price = S0*pow(u,c_base+l)*pow(d,1-(c_start-k)-c_base-l);
    float payoff = fmaxf(E-price, 0);
    dataA[l] = fmaxf(hold, payoff);
} k--;

7 test cases for GPU-implementation

Here, we use the same idea as we did when testing the CPU-implementation.

Kristoffer Fürst

Uppsala University
7.1 European put

The code is found in appendix C on page 16. To run the program, you call

\[ \text{> make euroGPU} \]
\[ \text{> ./euroGPU } S_0 \ T \ r \ vol \ E \]

where \( S_0 \) is the value of the underlying asset today, \( T \) is the time until expiry, \( r \) is the interest rate, \( vol \) is the volatility and \( E \) is the strike price.

The tests that follow are just a comparison between the CPU and GPU code. This is the most relevant test, since the codes are essentially the same, but calculated using different architectures.

### Table 3: These values will be used in the first four tests for the European put in this section.

<table>
<thead>
<tr>
<th>Interest rate (( r ))</th>
<th>0.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strike price (( E ))</td>
<td>10</td>
</tr>
<tr>
<td>Asset value (( S ))</td>
<td>5</td>
</tr>
<tr>
<td>Volatility (( \sigma ))</td>
<td>0.3</td>
</tr>
<tr>
<td>steps in time (NUM_STEPS)</td>
<td>1024</td>
</tr>
</tbody>
</table>

7.1.1 GPU European put, Test1

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td>4.43017</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>4.42992</td>
</tr>
</tbody>
</table>

7.1.2 GPU European put, Test2

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td>4.70452</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>4.70459</td>
</tr>
</tbody>
</table>

7.1.3 GPU European put, Test3

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td>4.85098</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>4.85096</td>
</tr>
</tbody>
</table>

7.1.4 GPU European put, Test4

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td>4.56348</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>4.56341</td>
</tr>
</tbody>
</table>
7.1.5 GPU European put, Test 5

Here, I compare to the exact value, and change a couple of parameters.

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>1</th>
<th>Interest rate (r)</th>
<th>0.05</th>
<th>Strike price (E)</th>
<th>100</th>
<th>Asset value (S)</th>
<th>100</th>
<th>Volatility (σ)</th>
<th>0.4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Exact value</th>
<th>13.15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>from CPU-code</td>
<td>13.1362</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>13.1369</td>
</tr>
</tbody>
</table>

7.1.6 GPU European put, Test 6

This test, as the previous, also compares the program’s result to an exact result, but with different parameters.

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.41096</th>
<th>Interest rate (r)</th>
<th>0.04</th>
<th>Strike price (E)</th>
<th>340</th>
<th>Asset value (S)</th>
<th>350</th>
<th>Volatility (σ)</th>
<th>0.2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Exact value</th>
<th>10.63</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>from CPU-code</td>
<td>10.7643</td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td>10.7625</td>
</tr>
</tbody>
</table>

7.1.7 Conclusion

We can see that the values are close to each other but not exactly the same. This is most likely due to the fact that machine-operations are not completely associative.

7.2 American put

The code is found in appendix D on page 18. To run the program, you call

```bash
> make amerGPU
> ./amerGPU S₀ T r vol E
```

where $S₀$ is the value of the underlying asset today, $T$ is the time until expiry, $r$ is the interest rate, $vol$ is the volatility and $E$ is the strike price.

The tests done here are the same as for the European put, i.e. simply a comparison between the value from the GPU-code and that of the CPU-code.
Table 4: These values will be used in all the tests for the American put in this section.

<table>
<thead>
<tr>
<th>Interest rate (r)</th>
<th>0.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strike price (E)</td>
<td>10</td>
</tr>
<tr>
<td>Asset value (S)</td>
<td>9</td>
</tr>
<tr>
<td>Volatility (σ)</td>
<td>0.3</td>
</tr>
<tr>
<td>steps in time (NUM_STEPS)</td>
<td>1024</td>
</tr>
</tbody>
</table>

7.2.1 GPU American put, Test 1

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td><strong>1.43423</strong></td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td><strong>1.4342</strong></td>
</tr>
</tbody>
</table>

7.2.2 GPU American put, Test 2

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td><strong>1.25519</strong></td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td><strong>1.25518</strong></td>
</tr>
</tbody>
</table>

7.2.3 GPU American put, Test 3

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td><strong>1.12525</strong></td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td><strong>1.12524</strong></td>
</tr>
</tbody>
</table>

7.2.4 GPU American put, Test 4

<table>
<thead>
<tr>
<th>Input</th>
<th>years until expiry (T)</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>from CPU-code</td>
<td><strong>1.35417</strong></td>
</tr>
<tr>
<td></td>
<td>from GPU-code</td>
<td><strong>1.35422</strong></td>
</tr>
</tbody>
</table>

7.2.5 Conclusion

Once again, the values are close but not exactly the same.

7.3 American call

As before, to evaluate a call option instead of a put option, the only thing that needs to be changed is the payoff function. This is a simple change in the code, and tests are again omitted.
8 Conclusion

The BOPM when implemented on a GPU provides *almost* the same result as when implemented on a CPU. The differences in result is most likely due to the fact that computer arithmetic is not completely associative.

To compare the speed between the CPU- and GPU-code one would need a better hardware setup than we used. This could be done by using GPUs and CPUs in the same price range and specialized for the same type of operations.

Another aspect to look at next could be how to incorporate dividend payments in the underlying asset. It is also possible to use the BOPM to value barrier options, which is a more complex kind of financial derivative.

A C++ code for European put

```cpp
#include <math.h>
#include <iostream>
using namespace std;

// Exercise (strike) price.
#define E 10

// Takes a value (S) of the underlying asset and returns the payoff for a put.
float payoff_put(float a) {
    float payoff = E - a;
    float b = 0.0f;
    return (payoff > b) ? payoff : b;
}

// Evaluates a European put with respect to given parameters.
void euro_option(float S0, float u, float d, float p, float r, int NUM_STEPS, float dt) {
    // Used to store the values of the asset, and also the values of the option.
    float results[NUM_STEPS + 1];

    // Value of the option at expiry, based on the value of the value of the asset at expiry (the option can have #NUM_STEPS+1 values at expiry).
    for (int i = 0; i <= NUM_STEPS; i++) {
        results[i] = S0 * pow(u, i) * pow(d, NUM_STEPS - i); // Value of asset at this leaf.
        results[i] = payoff_put(results[i]); // Value of option at this leaf.
    }

    // Find the value of the option at all times, working back up the tree.
    for (int i = NUM_STEPS; i > 0; i--){
        for (int j = 0; j < i; j++){
            results[j] = expf(-r * dt) * (p * results[j + 1] + (1 - p) * results[j]);
        }
    }

    cout << results[0] << endl; // Output is the value of the option today.
}
```

```cpp
int main ( int argc , char * argv[])
{
    if (argc < 5) {
        cerr << "Syntax: S0, T, r, vol, NUM_STEPS" << endl;
        return 1;
    }
    float S0, T, r, vol, NUM_STEPS;
    S0 = atof(argv[1]);
    T = atof(argv[2]);
    r = atof(argv[3]);
    vol = atof(argv[4]);
    NUM_STEPS = atoi(argv[5]);

    euro_option(S0, exp(r * T), 1 / (1 + vol), r, vol, NUM_STEPS, T / NUM_STEPS);
    cout << results[0] << endl;
}
```
exit(0);
}

float S0(atof(argv[1]));  // Current value of asset.
float T(atof(argv[2]));    // Time (in years) until expiry.
float r(atof(argv[3]));    // Risk-free interest rate.
float vol(atof(argv[4]));  // Volatility.
int NUM_STEPS(atoi(argv[5]));// Number of time steps.
float dt = T/NUM_STEPS;    // Size of each time step.

float A = 0.5*(expf(-r*dt)+expf((r+vol)*vol)*dt));  // Intermediate
float d = A-sqrt(A*A-1);  // Coefficient of asset value when moving up.
float u = A+sqrt(A*A-1);  // Coefficient of asset value when moving down.
float p = (expf(r*dt)-d)/(u-d);  // Probability of moving up.

euro_option(S0, u, d, p, r, NUM_STEPS, dt); // Evaluate the option.

B  

C++ code for American option

```cpp
#include <iostream>
#include <math.h>
using namespace std;

// Exercise (strike) price.
#define E 10

// Takes a value (S) of the underlying asset and returns the payoff for a put.
float payoff_put(float S){
    float ret = E-S;
    float b = 0.0f;
    return fmaxf(ret, b);
}

// Takes a value (S) of the underlying asset and returns the payoff for a call.
float payoff_call(float S){
    float ret = S-E;
    float b = 0.0f;
    return fmaxf(ret, b);
}

// Evaluates an American option with respect to given parameters.
void amer_option(float S0,
    float u, 
    float d, 
    float p, 
    float r, 
    int NUM_STEPS, 
    float dt, 
    string type)
{
    // Used to store the values of the asset, and also the values of the option.
    float results[NUM_STEPS+1];
    // Value of the option at expiry, based on the value of the value of the
    // asset at expiry (the option can have #NUM_STEPS+1 values at expiry).
    for(int i = 0; i < NUM_STEPS; i++){
        results[i] = S0*pow(u,i)*pow(d,NUM_STEPS-i);  // Value of asset at this leaf.
        results[i] = payoff_put(results[i]);  // Value of option at this leaf.
    }
    // Find the value of the option at all times, working back up the tree. At
    // every node in the tree, the value is either the value of the option if
```
// it is kept or the value if the option is exercised and the money put in
// a bank account.
for (int i = NUM_STEPS; i >= 0; i --) {
  for (int j = 0; j <= i + 1; j++) {
    float hold = expf(-r*dt)*p*results[j+1]+(1-p)*results[j]);
    const char *p = type.c_str();
    // If the user has chosen a put–option
    if (strcmp("put", p) == 0) {
      float payoff = payoff_put(S0*expf(u*j)*expf(d*(i-j)));
      results[j] = max(hold, payoff);
    }
    // If the user has chosen a call–option.
    else if (strcmp("call", p) == 0) {
      float payoff = payoff_call(S0*expf(u*j)*expf(d*(i-j)));
      results[j] = max(hold, payoff);
    }
    // If wrong input has been used.
    else {
      cerr << "type has to be 'put' och 'call'" << endl;
      exit(0);
    }
  }
}
cout << results[0] << endl; // Output is the value of the option today.

int main (int argc, char *argv[]) {
  if (argc < 6) {
    cerr << "Syntax: S0, T, r, vol, NUM_STEPS, type" << endl;
    exit(0);
  }
  float S0 = atof(argv[1]); // Current value of asset.
  float T = atof(argv[2]); // Time (in years) until expiry.
  float r = atof(argv[3]); // Risk-free interest rate.
  float vol = atof(argv[4]); // Volatility.
  int NUM_STEPS = atof(argv[5]); // Number of time–NUM_STEPS.
  string type = argv[6]; // Type of option.
  float dt = T/NUM_STEPS; // Size of each time–step.
  float p = 0.5; // probability of moving up.

  // Koefficient of asset–value when moving up.
  float d = expf(r*dt)*sqrtf(expf(2*vol*vol*dt)-1));
  // Koefficient of asset–value when moving down.
  float u = expf(r*dt)*sqrtf(expf(2*vol*vol*dt)-1));
  amer_option(S0, u, d, p, r, NUM_STEPS, dt, type); // Evaluate the option.
  system("PAUSE");
  return 0;
}

C CUDA code for European put

#include <iostream>

Kristoffer Fürst

Uppsala University
#include <math.h>
#include <cutil.h>
using namespace std;

#define CACHE_STEP 512 // Number of threads, must be multiple of CACHE_DELTA
#define CACHE_DELTA 32 // Number of steps back in time per loop.
#define CACHE_SIZE (CACHE_STEP + CACHE_DELTA)
#define NUM_STEPS 1024 // Number of time-steps, must be multiple of CACHE_DELTA
#define GRID_N 1024 // Number of blocks with threads

// Used to store the values of the asset, and also the values of the option.
static __device__ float results[NUM_STEPS+1];

// Evaluates a European put with respect to given parameters.
__global__ void euro_option(float S0,
float u,
float d,
float p,
float r,
float dt,
float sd_value,
float vol)
{
    // Memory shared between threads in a block
    __shared__ float dataA[CACHE_SIZE];
    __shared__ float dataB[CACHE_SIZE];

    // Value of the option at expiry, based on the value of the value of the asset at expiry (the option can have # (steps + 1) values at expiry).
    for(int i = threadIdx.x; i <= NUM_STEPS ; i += blockDim.x){
        float price = S0 * pow(u, i) * pow(d, NUM_STEPS-i);
        results[i] = fmax(E-price, 0);
    }

    // Find the value of the option at all times, working back up the tree.
    for(int i = NUM_STEPS; i >= 0; i -= CACHE_DELTA){
        for(int c_base = 0; c_base <= i+1; c_base += CACHE_STEP){
            // Start and end positions within shared memory cache
            int c_start = min(CACHE_SIZE - 1, i - c_base);
            int c_end = c_start - CACHE_DELTA;

            // Read data (with apron) to shared memory cache
            __syncthreads();
            for(int k = threadIdx.x; k <= c_start; k += blockDim.x)
                dataA[k] = results[c_base + k];

            // Calculations within shared memory
            for(int k = c_start - 1; k >= c_end;)
            {
                // Compute discounted expected value
                __syncthreads();
                for(int l = threadIdx.x; l <= k; l += blockDim.x)
                {
                }
                k--;

                // Compute discounted expected value
                __syncthreads();
                for(int l = threadIdx.x; l <= k; l += blockDim.x)
                {
                }
                k--;
            }
        }
    }

    // Flush shared memory cache
    __syncthreads();
    for(int k = threadIdx.x; k <= c_end; k += blockDim.x)
results[c_base + k] = dataA[k];

// Write the value at the top of the tree to destination buffer
if (threadIdx.x == 0) *d_value = results[0];

int main (int argc, char * argv[])
{
    if (argc < 5) {
        cerr << "Syntax: S0 T r vol E" << endl;
        exit(0);
    }

    float S0(atof(argv[1]));   // Current value of asset.
    float T(atof(argv[2]));     // Time (in years) until expiry.
    float r(atof(argv[3]));     // Risk-free interest rate.
    float vol(atof(argv[4]));   // Volatility.
    float E(atof(argv[5]));     // Strike (exercise) price
    float sd_value;             // Value today on device
    float sh_value;             // Value today on host
    float dt = T/NUM_STEPS;     // Size of each time-step.

    float A = 0.5*(expf(-r*dt)+expf((r+vol*vol)*dt));
    float d = A-sqrt(A*A-1);   // Koefficient of asset-value when moving down.
    float u = A+sqrt(A*A-1);   // Koefficient of asset-value when moving up.
    float p = (expf(r*dt)-d)/(u-d);  // Probability of moving up.

    h_value = (float*)malloc(sizeof(float));
    CUDA_SAFE_CALL( cudaMalloc((void **)&d_value, sizeof(float)) );
    CUDA_SAFE_CALL( cudaThreadSynchronize() );

    dim3 blockDim(ceil(NUM_STEPS/GRID_N),1);   //Number of blocks.
    dim3 gridSize(GRID_N,1);                   //Number of threads per block
    euro_option<<<block, grid>>>(S0, E, u, d, p, r, dt, d_value, vol);

    CUT_CHECK_ERROR("euro_option() execution failed\n");
    CUDA_SAFE_CALL( cudaMemcpy(h_value, d_value, sizeof(float),
                                 cudaMemcpyDeviceToHost) );

    cout << *h_value << endl;
    CUDA_SAFE_CALL( cudaFree(d_value) );
    free(h_value);

    return 0;
}

D CUDA code for American put

/*@ U.S. Government End Users. This source code is a "commercial item" as
that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting of
"commercial computer software" and "commercial computer software
documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
and is provided to the U.S. Government only as a commercial end item.
Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202–1 through
227.7202–4 (JUNE 1995), all U.S. Government End Users acquire the
source code with only those rights set forth herein.*/

#include <iostream>
#include <math.h>
#include <util.h>
using namespace std;

#define CACHE_STEP 128  //Number of threads, must be multiple of CACHE_DELTA
#define CACHE_DELTA 16  //Number of steps back in time per loop.

Kristoffer Fürst
18
Uppsala University
#define CACHE_SIZE (CACHE_STEP + CACHE_DELTA)
#define NUM_STEPS 1024 // Number of time-steps, must be multiple of CACHE_DELTA
#define GRID_N 512 // Number of blocks with threads

static __device__ float results[NUM_STEPS+1]; // Values of the option

// Evaluates a European put with respect to given parameters.
__global__ void amer_option(float S0,
  float E, float u, float d, float p, float r, float dt, float wd_value, float vol)
{

  // Memory shared between threads in a block
  __shared__ float dataA[CACHE_SIZE];
  __shared__ float dataB[CACHE_SIZE];
  results[0] = S0;

  // Value of the option at expiry, based on the value of the value of the
  // asset at expiry (the option can have # steps+1 values at expiry).
  for(int i = threadIdx.x; i <= NUM_STEPS ; i += blockDim.x)
  {
    float price = S0*exp(u*i)*exp(d*(NUM_STEPS-i));
    results[i] = fmax(E-price, 0);
  }

  // Find the value of the option at all times, working back up the tree.
  for(int i = NUM_STEPS; i > 0; i -= CACHE_DELTA)
  for(int c_base = 0; c_base <= 1; c_base += CACHE_STEP)
  {
    // Start and end positions within shared memory cache
    int c_start = min(CACHE_SIZE - 1, i - c_base);
    int c_end = c_start - CACHE_DELTA;

    // Read data (with apron) to shared memory cache
    __syncthreads();
    for(int k = threadIdx.x; k <= c_start; k += blockDim.x)
      dataA[k] = results[c_base + k];

    // Calculations within shared memory
    for(int k = c_start - 1; k >= c_end;)
    {
      // Compute discounted expected value
      __syncthreads();
      for(int l = threadIdx.x; l <= k; l += blockDim.x)
      {
        float hold = expf(-r*dt)*(p*dataA[l+1]+(1-p)*dataA[l]);
        float price = S0*exp(u*c_base+l)*exp(d,i-(c_start-k)-
          c_base-l); 
        float payoff = fmax(E-price, 0);
        dataB[l] = fmax(hold, payoff);
      }
      k--;

      // Compute discounted expected value
      __syncthreads();
      for(int l = threadIdx.x; l <= k; l += blockDim.x)
      {
        float hold = expf(-r*dt)*(p*dataB[l+1]+(1-p)*dataB[l]);
        float price = S0*exp(u,c_base-l)*exp(d,i-(c_start-k)-
          c_base-l); 
        float payoff = fmax(E-price, 0);
        dataA[l] = fmax(hold, payoff);
      }
      k--;
    }

    // Flush shared memory cache
    __syncthreads();
  }

Kristoffer Fürst
19 Uppsala University
for(int k = threadIdx.x; k <= c_end; k += blockDim.x)
    results[c_base + k] = dataA[k];
}
//Write the value at the top of the tree to destination buffer
if(threadIdx.x == 0) *d_value = results[0];

int main ( int argc, char * argv[] )
{
    if (argc < 5) {
        cerr << "Syntax: S0 T r vol E" << endl;
        exit(0);
    }
    float S0( atof(argv[1])); //Current value of asset.
    float T( atof(argv[2])); //Time (in years) until expiry.
    float r( atof(argv[3])); //Risk-free interest rate.
    float vol( atof(argv[4])); //Volatility.
    float E( atof(argv[5])); //Strike (exercise) price.
    float *d_value;
    //Value today on device.
    float *h_value;
    //Value today on host.
    float dt = T/NUM_STEPS; //Size of each time-step.
    float p = 0.5f; //Probability of moving up.
    float d = expf(r*dt)*(1-sqrtf(expf(vol*vol*dt)-1)); //Koefficient of asset-value when moving down.
    float u = expf(r*dt)*(1+sqrtf(expf(vol*vol*dt)-1)); //Koefficient of asset-value when moving up.
    float h_value = malloc(sizeof(float));
    CUDA_SAFE_CALL(cudamalloc((void **)&d_value, sizeof(float)));
    CUDA_SAFE_CALL(cudathreadSynchronize());
    dim3 block(ceil(NUM_STEPS/GRID_N),1); //Number of blocks
    dim3 grid(GRID_N,1); //Number of threads per block
    amer_option<<<block, grid>>>(S0, E, u, d, r, dt, d_value, vol);
    CUT_CHECK_ERROR("euro_option() execution failed\n");
    CUDA_SAFE_CALL( cudaMemcpy(h_value, d_value, sizeof(float), cudaMemcpyDeviceToHost) );
    cout << "h_value << endl;
    CUDA_SAFE_CALL( cudaFree(d_value) );
    free(h_value);
    return 0;
}

References