Extended Abstract: A Functional Approach to Monte Carlo based American Option Pricing

Wojciech Michal Pawlak  
Department of Computer Science,  
University of Copenhagen  
SimCorp Technology Labs  
wmp@di.ku.dk, wmpk@simcorp.com

Martin Elsman  
Department of Computer Science,  
University of Copenhagen  
mael@di.ku.dk

Cosmin Eugen Oancea  
Department of Computer Science,  
University of Copenhagen  
cosmin.oancea@di.ku.dk

Abstract
We study the feasibility and performance efficiency of expressing a complex financial numerical algorithm with high-level functional parallel constructs. The algorithm we investigate is a least-square regression-based Monte-Carlo simulation for pricing American options. We propose an accelerated parallel implementation in Futhark, a high-level functional data-parallel language. The Futhark language targets GPUs as the compute platform and we achieve a performance comparable to an implementation optimised by NVIDIA CUDA engineers. In absolute terms, we can price a put option with 1 million simulation paths and 100 time steps in 20ms on a NVIDIA Tesla V100 GPU.

CCS Concepts  • Computing methodologies → Shared memory algorithms; Massively parallel algorithms; Massively parallel and high-performance simulations; Parallel programming languages; • Applied computing → Economics; • Computer systems organization → Multicore architectures;

Keywords High-Performance Computing, Parallel (GPU) Programming, Functional Programming, Compilers, Computational Finance, Derivative Pricing

ACM Reference Format:

1 Introduction
Pricing American options is a fundamental business case in the financial services sector as such financial instruments are widely traded in the derivative markets. American options can be exercised at any time between the present date and the time to maturity. This aspect puts them in contrast to European options, that can only be exercised at their maturity. In the usual case, the option holder is expected to exercise the option as soon as it is more profitable to do so rather than wait until its expiration. Effectively, the value of an American option is the value achieved by exercising optimally. This embedded optimisation (optimal stopping) problem is the main challenge. As there is no general closed-form formula solutions [17], it is necessary to approximate the option value accurately with a numerical simulation. This is a substantial and timely computational effort. It has to be significantly reduced to become acceptable for time-critical applications in financial practice. Therefore it is a valid case for performance acceleration through massive parallelisation on GPUs.

Current most efficient accelerated simulations are implemented in dedicated languages and frameworks like CUDA [1, 15, 27], MPI [10], OpenMP [32], and other technologies [9]. The challenge with these implementations is the poor expressibility, which makes them inaccessible to domain experts and limits them to the specialist developers. It also results in code that is difficult to maintain. On top of that, one needs to be aware of the low-level properties of the underlying hardware architecture.

We propose a functional approach to the implementation of an accelerated option pricing model. The use of high-level parallel constructs lets us express the algorithm in an intuitive manner, without the implementation concerns of mapping the code to the architecture. The Futhark language and the optimizing compiler behind it make this possible [21]. Previous work has investigated the use of Futhark for implementing Monte Carlo based European option pricing [2], which covered a number of advanced features that the present work does not consider, including Sobol sequence generation [18]. Quite a few approaches exist aiming at generating efficient data-parallel GPGPU code for applications written using high-level array language constructs, including the work on Obsidian [11, 29, 30] and Accelerate [8], which are both domain specific languages embedded in Haskell, but, which do not feature arbitrary nested parallelism. Approaches that support arbitrary nested parallelism includes the seminal work on flattening of nested parallelism in NESL [4, 5], which was extended to operate on a richer set of values in Data-parallel Haskell [7], and the work on data-only flattening [33]. However, such general compiler-based flattening is challenging to implement efficiently on GPUs [3].

Other promising attempts at compiling NESL to GPUs include Nessie [28], which is still under development, and CuNesl [33], which aims at mapping different levels of nested
parallelism to different levels of parallelism on the GPU, but which lacks critical optimisations such as fusion.

The main contribution of this work is an efficient Futhark implementation of a pricing model for American options, a Longstaff-Schwartz algorithm using Monte Carlo Simulation with Least-Square Regression (abbreviated LSMC) [25]. We present our implementation and compare the performance results to a benchmark CUDA version. We obtain results that are on par with a hand-tuned version written in a dedicated low-level programming model.

2 Design and Implementation

Futhark is a statically typed parallel functional array language. The language is based on an ML or Haskell style syntax and is equipped with a number of second-order array combinators (SOACs), such as map, reduce, scan, and filter. The Futhark language features a higher-order module system [14], polymorphism, and a restricted form of higher-order functions [23], concepts that are all eliminated at compile time and introduce no overhead at runtime. The Futhark language is equipped with a number of second-order ar

The computational effort of a Monte Carlo simulation is determined by the number of simulation paths and time steps. A large number of paths $n$, usually 100,000 to 1,000,000, needs to be generated to obtain an accurate value approximation [16]. The number of time steps $m$ is bound to the number of early-exercise opportunities and is usually much smaller than $n$. We use a minimum standard pseudo-random number generator in a parallel skip-ahead fashion.

The backward dynamic programming steps 4a and 4b in the intrinsically sequential loop are the main performance bottlenecks as we need to (1) perform operations on a matrix of size $n \times 3$ in the worst case, (2) deal with matrix sizes that vary across steps (thread divergence), and (3) make sure that threads are synchronised after each step. The payoff function determines the number of relevant (in-the-money) paths at each time step, and thus the total computational effort of this part. We therefore hoist this computation out of the loop and prepare small fixed-size matrices by performing the computation in parallel across time steps. We achieve this goal through a chain of linear algebra transformations, such as SVD and QR decomposition as well as a pseudo-inverse transform. In addition, we specialise the algorithm to work with a 3-degree monomial basis function. We do not claim any contributions to this algorithm. In fact, for this part, our implementation closely follows the implementation proposed by NVIDIA [13, 26]. We focus on the goal to match the performance of this publicly available benchmark implementation. The algorithm outlined in Listing 1 is implemented using a nested composition of sequential loops and parallel map, reduce, and scan constructs.

3 Experimental Results

We have run experiments on a Linux system with a 26-core 2-way HT Intel Xeon Platinum 8167M CPU (2.00GHz), 754 GB DDR RAM and NVIDIA Tesla V100 SXM2 GPU (2688...
is Futhark compiled to CUDA. Both total and partial American option pricing. We provide an option price obtained from a different numerical method (binomial tree) for reference. We use it to validate the result.

Volta FP64 cores, 16 GB HBM2 using CUDA 10.1. The pricing test case is presented in Table 1. The performance results are presented in Table 2.

<table>
<thead>
<tr>
<th>Model Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option Type, Payoff</td>
<td>Put, max(K − S)</td>
</tr>
<tr>
<td>Initial spot price (S0)</td>
<td>80.0</td>
</tr>
<tr>
<td>Strike price (K)</td>
<td>90.0</td>
</tr>
<tr>
<td>Time to maturity (T)</td>
<td>1 year</td>
</tr>
<tr>
<td>Risk free rate (r)</td>
<td>5%</td>
</tr>
<tr>
<td>Volatility (σ)</td>
<td>30%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time steps/Early Exercise dates</td>
<td>100</td>
</tr>
<tr>
<td>Paths</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Option Price (Binomial Tree)</td>
<td>13.804</td>
</tr>
</tbody>
</table>

Table 1. Set of model and simulation parameters for the American option pricing. We provide an option price obtained from a different numerical method (binomial tree) for reference. We use it to validate the result.

Table 2. Execution times for the test case. Ref is the CUDA benchmark, while V1 is Futhark compiled to OpenCL and V2 is Futhark compiled to CUDA. Both total and partial execution times for each part of the algorithm are shown. Execution times are given in ms. The runtimes are averaged based on 250 runs. We compare the speedups against the slowest runtime.

<table>
<thead>
<tr>
<th>Path</th>
<th>SVD</th>
<th>Main</th>
<th>Total</th>
<th>Speedup</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>4.7</td>
<td>1.8</td>
<td>8.9</td>
<td>15.4</td>
<td>1.45× 13.778</td>
</tr>
<tr>
<td>V1</td>
<td>5.9</td>
<td>2.1</td>
<td>14.3</td>
<td>22.3</td>
<td>1.00× 13.789</td>
</tr>
<tr>
<td>V2</td>
<td>5.4</td>
<td>1.9</td>
<td>13.1</td>
<td>20.4</td>
<td>1.09× 13.789</td>
</tr>
</tbody>
</table>

References


