A Functional Approach to Accelerating Monte Carlo based American Option Pricing

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

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

Cosmin Eugen Oancea
Department of Computer Science,
University of Copenhagen
Denmark
cosmin.oancea@diku.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, and in particular cases up to 2.5× better than, 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 17 ms on a NVIDIA Tesla V100 GPU. Furthermore, the high-level functional specification is much more accessible to the financial-domain experts than the low-level CUDA code, thus promoting code maintainability and facilitating algorithmic changes.

CCS CONCEPTS

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

KEYWORDS

High-Performance Computing, Parallel (GPU) Programming, Functional Programming, Computational Finance

ACM Reference Format:
Wojciech Michal Pawlak, Martin Elsman, and Cosmin Eugen Oancea. 2020. A Functional Approach to Accelerating Monte Carlo based American Option Pricing . In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Pricing American options is a fundamental business case in the financial services sector, because 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, which can be exercised only 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 it at the optimal time. This embedded optimal stopping problem is a challenge, for which there is no general closed-form formula [28]. Instead, the option value must be approximated with a numerical simulation, which constitutes a substantial computational effort that must be executed efficiently to be acceptable for time-critical financial applications.

Currently, the most efficient accelerated simulations are implemented in dedicated languages and frameworks such as CUDA [1, 24, 43], OpenCL [10, 53], MPI [16], OpenMP [54] and other technologies [15]. A challenge with such implementations is the poor expressibility, which makes them inaccessible to domain experts and difficult to maintain. Specialist developers must be appointed to implement and maintain the code, which require a deep understanding of the underlying hardware and experience with code transformations aimed at optimising potentially-conflicting factors such as locality of reference and the degree of parallelism.

We propose a functional approach to implementing an accelerated option pricing model. The use of high-level parallel constructs lets us express the algorithm in an intuitive manner, without the concerns of mapping the code to the architecture. The Futhark language and the optimising compiler behind it makes this approach possible [33]. The paper makes the following contributions:

(1) We present a high-level data-parallel implementation of the Longstaff-Schwartz algorithm for pricing American options using Monte Carlo Simulation with Least-Square Regression (abbreviated LSMC) [37]. The implementation serves as a reference implementation and can easily be ported to other languages. Moreover, the algorithm makes explicit the available parallelism by using high-level data-parallel constructs.
(2) We give a detailed description of the algorithmic changes required to achieve an efficient parallel implementation of the LSMC algorithm.
(3) We present an optimised efficient version of the algorithm and describe how the original algorithm is rewritten in

Futhark to achieve performance results that in most cases matches a CUDA version, which is a hand-tuned implementation of the algorithm, written in a dedicated low-level programming model (by CUDA engineers). In addition, we present specific cases, in which the Futhark version achieves up to 2.5× speedup over the CUDA version.

The remainder of the paper is organised as follows. In Section 2, we examine the problem solved by our application and describe the high-level algorithm for American option pricing. In Section 3, we describe the Futhark language and show how American option pricing is implemented in Futhark. In Section 4, we describe and demonstrate how the inefficient version of the algorithm is turned into an efficient version targeting GPUs. Section 5 presents and discusses the main experimental results. Section 6 reviews related work and Section 7 summarises our main findings.

2 AMERICAN OPTION PRICING

An option contract is defined by its payoff function. For the vanilla-type options like *calls* and *puts*, it compares the strike price K and the current asset spot price S and thereby determines the cash flow of an option. The strike price K is an agreed fixed price, at which the option holder can buy (in case of a call) or sell (in case of a put) the underlying asset. The spot price S is a price of the underlying asset like stock or commodity, that the option derives its value from. S varies over time, which can be described using a stochastic process. Such a process is defined by a stochastic differential equation (SDE), which cannot be solved directly using a closed-form formula. Instead, we use a numerical simulation method.

In practice, Monte Carlo (MC) simulation is the most widely used and robust method for solving general SDE problems, and option pricing problems in particular. It is popular in the quantitative finance community, because (1) it is relatively simple to implement (and parallelise) and (2) it allows for pricing of complex instruments that depend on many underlying assets. Such instruments cannot be priced with deterministic methods such as finite difference or lattice models suitable for low-dimensional instruments. In fact, Monte Carlo is the only numerical method that can be used for multi-factor pricing in dimensions greater than four [25].

The Monte Carlo Simulation method is based on two theorems of probability theory. The first one is the *Strong Law of Large Numbers*, which guarantees the convergence of a certain series of independent random numbers having the same distribution to a value of an integral. The second is the *Central Limit Theorem*, which determines the convergence rate of the first law [4, 48]. In a standard Monte Carlo simulation, the paths of the state variables are simulated forward in time, which is, in particular, the case for pricing European options. Given an option payoff, a forward (future) price is determined for each path at the maturity date. To estimate the present price of the instrument, the future cash flows (prices in different points in future time) have to be discounted to the present time using some established discount interest rate (i.e., money available now is worth more than the identical amount in the future.) Finally, an unbiased estimate of the current option price is a mean average of these prices.

In contrast to pricing European options, the pricing of American options happens backward in time. First, the optimal exercise price is determined at maturity and, subsequently, is recursively propagated and discounted backward in time (until the current time) using dynamic programming. Although an American option can be exercised at any time, the exercise times are discretised and restricted to a fixed set of times (e.g., daily or monthly). The overall goal here is to provide an approximation of the optimal stopping rule that maximises the value of the American option.

At maturity $t = T$, the option holder will exercise the option if it is *in-the-money* (ITM), that is, if the value of the option, if exercised, will generate a positive cash flow (e.g., in case of a call option, a spot price higher than the strike price is preferable.) For any other time t_i, the holder needs to choose whether to exercise the option or continue to hold it. The option value is at the maximum if the exercise happens as soon as the immediate exercise cash flow is greater than or equal to the continuation value (i.e., the discounted expected option value at the next instance in time.) However, this continuation value at any given time t_i is not known, so it needs to be estimated.

In practice, we want to price large portfolios of options through simulation within seconds for real-time decision making. That is why GPUs, which allow for a high degree of parallelism due to its massive number of cores, are a good fit for such workloads.

2.1 The Longstaff-Schwartz Algorithm

Several authors have proposed the use of regression to estimate continuation values from simulated paths and thereby enable American option pricing through simulation [11, 37, 52]. There are several reasons for choosing the Longstaff-Schwartz approach [37]. First, it is the most widely adopted algorithm for American option pricing in the financial industry. Second, it is based on an easily-parallelisable Monte Carlo simulation. Finally, the convergence of the Longstaff-Schwartz algorithm have been widely studied together with its dependence on the number of simulated paths [18].

2.1.1 *Least-Square Regression.*

To start with, we turn our focus to the core challenge of the algorithm—the estimation of the continuation values C_i at each time step i. Let us assume that S_{ij} is an asset spot price at time step i on a path j. Each continuation value $C_i(S_{ij})$ is the regression of the option value in the next time step on the value of S_{ij} in the current time step and path. The procedure is to approximate C_i by a linear combination of basis functions of the current state and use regression to estimate the best coefficients for this approximation. The approximation accuracy depends on the choice of functions used in the regression.

Following the Longstaff-Schwartz approach, we apply an ordinary least-squares regression across the simulated paths at any given time t_i to estimate the continuation value C_i. The least-square regression is a method for finding approximate solutions of over-determined[1] systems of linear equations by minimising the sum of the squares of the errors in the equations [9]. However, since the decision to exercise the option is relevant only when the option is in-the-money, we regress only paths that are in-the-money. This choice results in an improved algorithmic efficiency without negative impact on accuracy.

[1] There are more equations to solve than variables to choose.

To begin with, the ITM paths are parameterised using a quadratic polynomial $\beta_0 + \beta_1 S_{ij} + \beta_2 S_{ij}^2$, where S_{ij} is a variable—an asset spot price at a time step i, here for some ITM path j. In particular, each term of a polynomial, a monomial with basis $1, x, x^2, \dots$, is a basis function. We chose to use up to second order polynomials in the basis. Having said that, we mention that this choice is usually left as a parameter to the algorithm. In contrast, we deliberately settle on a particular type of basis function. This makes it possible to implement specialised versions of matrix transformations, and as a result enable performance optimisations. The number of used polynomials matches the number of different time steps i, because one regression is performed for each step. A point cloud of asset spot prices S_ij, distributed for each path across the time steps, is obtained from a simulation. The regression problem is then to find, separately for each of these time steps, the best fit in terms of β coefficients and basis functions for a quadratic polynomial. We discuss this procedure further in Section 4.

Essentially, we deal with an over-determined system of equations, because the number of equations is larger than the number of unknown coefficients (i.e., 3β coefficients). For the sake of presentation, let us assume we solve a system of equations of form $Ax = b$. We minimise the objective function $\| Ax - b \|^2$ by finding a variable vector $\hat{x}$ from all possible choices of x. In other words, it is a least squares approximate solution to $\| A\hat{x} - b \|^2 \leqslant \| Ax - b \|^2$. In our case, we assume that the number of ITM paths itm is significantly larger than the number of the basis functions (3). It follows that A is a tall data matrix of size $itm \times 3$ and b is a data column vector of size itm. The variable of this system of equations is the column vector x of size 3. In particular, A is a matrix of powers of asset spot prices S_i built from the chosen polynomial. b is a vector of cash flows $\hat{V}_i$ dependent on the payoff function $p(S_i)$, specific for an option that is priced. Finally, x is a vector of polynomial coefficients β_k, that we want to fit with the least squares method. We arrive at the following system of equations:

$$\begin{bmatrix} 1 & S_0 & S_0^2 \\ & \dots & \\ 1 & S_j & S_j^2 \\ & \dots & \\ 1 & S_{itm-1} & S_{itm-1}^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} p(S_0) \\ \dots \\ p(S_j) \\ \dots \\ p(S_{itm-1}) \end{bmatrix} \quad (1)$$

The textbook solution for solving this least-square problem is to multiply both sides by A^T, resulting in $A^T A\hat{x} = A^T b$. Since $A^T A$ is a square matrix, we can now multiply both sides with its inverse[2] resulting in the unique solution: $\hat{x} = (A^T A)^{-1} A^T b$. The matrix $(A^T A)^{-1} A^T$ is the pseudo-inverse of the matrix A, denoted $A^\dagger$. The approach to constructing $A^\dagger$ is the main algorithmic challenge and the source for optimisations. There exist different methods to build $A^\dagger$. For the first naive implementation, we use the formula directly, applying matrix multiplication, transpositions, and inversion on A. In Section 4.2, we present an efficient algorithm for a pseudo-inverse construction, adapted for massive parallelism on GPUs.

2.1.2 *Algorithmic Structure.*

The generic structure of a simulation algorithm that employs a linear least squares regression according to the Longstaff-Schwartz algorithm [25] is summarised as follows.

[2] A fundamental assumption of least squares method is that the columns of matrix A are linearly independent, therefore $A^T A$ is always invertible.

We assume we are given the number of time steps m, the number of paths n, and the option-specific payoff function $p_i(S_{ij})$, where S_{ij} is an asset spot price at time step i and path j. Payoff p_i is discounted back from the maturity T (time step $m-1$) to current time (time step i).

A generic linear combination of basis functions is denoted by a polynomial function $\psi_i : \mathbb{R}^r \to \mathbb{R}$ and constant coefficients β_{ik}, where r is the highest degree of the chosen polynomial with each term being a basis function $k = 0, \dots, r-1$. Moreover, $\beta_i = [\beta_{i0}, \dots, \beta_{im-1}]$ and $\psi(S_i)^T = [\psi_0(S_i), \dots, \psi_{m-1}(S_i)]^T$.

(1) Generate a matrix $W(n, m)$ of random numbers drawn from a standard normal distribution.
(2) Using W, simulate (by *forward induction*) n independent paths $S_{0j}, \dots, S_{m-1j}, j = 0, \dots, n-1$ of Geometric Brownian Motion stochastic processes for the underlying asset prices.
(3) At the last step $m-1$ (at maturity T), compute the option value $\hat{V}_{mj} = p_m(S_{mj}), j = 0, \dots, n-1$ applying the payoff function p at the last step $m-1$.
(4) Apply *backward induction* for each step $i = m-2, \dots, 1$ to compute cash flows:
  (a) Select the ITM paths.
  (b) Build the matrix ψ_i from asset prices S_i and the right hand side cash flows vector $\hat{V}_{i+1}$ only for the ITM paths for the least-square linear equation $\psi_i(S_i)\beta_i = \hat{V}_{i+1}(S_{i+1})$.
  (c) Use regression to calculate $\hat{\beta}_i$ by solving a pseudo-inverse
  $$\psi(S_i)^\dagger = (\psi(S_i)^T \psi(S_i))^{-1} \psi(S_i)^T$$
  in
  $$\hat{\beta}_i = \psi(S_i)^\dagger \hat{V}_{i+1}(S_{i+1}).$$
  (d) Approximate the continuation function $\hat{C}_i(S_i) = \hat{\beta}_i \psi(S_i)^T$.
  (e) Decide to early-exercise based on the value of the continuation function $\hat{C}_i$ for each ITM path j:
  $$\hat{V}_{ij} = \begin{cases} p_i(S_{ij}), & p_i(S_{ij}) \geqslant \hat{C}_i(S_{ij}); \\ \hat{V}_{i+1,j}, & p_i(S_{ij}) < \hat{C}_i(S_{ij}). \end{cases} \quad (2)$$
(5) Return $\hat{V}_0 = (\hat{V}_{10} + \dots + \hat{V}_{1n-1})/n$ discounted to time step $i = 0$.

3 THE FUTHARK LANGUAGE

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 surface language features a higher-order module system [21], polymorphism, and limited support for higher-order functions [35].[3] Just as higher-order functions and modules are eliminated entirely at compile time, using a no-overhead approach, arrays of records (or tuples) are turned into records of flat arrays. The language also features a uniqueness type system and explicit sequential `loop` constructs, which, together with support for array-updates, allows for implementing imperative-like algorithm in a functional style.

The Futhark compiler supports aggressive fusion of parallel constructs [32], and specialised code generators for key parallel operators, such as map-scan and (segmented) map-reduce compositions [31, 36]. Essentially, since all available parallelism is assumed

[3] Functions may not be stored in arrays or returned by conditional expressions.

```
iota n = [0,...,n−1]
replicate n v = [v, ..., v] -- v repeated n times
map f [$a_1$,...,$a_n$] = [f $a_1$,..., f $a_n$]
map2 g [$a_1$,...,$a_n$][$b_1$,...,$b_n$] = [g $a_1$ $b_1$,...,g $a_n$ $b_n$]
reduce ⊙ $e_⊙$ [$a_1$,...,$a_n$] = $e_⊙$ ⊙ $a_1$ ⊙...⊙ $a_n$
$scan^{inc}$ ⊙ $e_⊙$ [$a_1$,...,$a_n$] = [$a_1$,..., $a_1$ ⊙...⊙$a_n$]
$scan^{exc}$ ⊙ $e_⊙$ [$a_1$,...,$a_n$] = [$e_⊙$,$a_1$,...,$a_1$ ⊙...⊙$a_{n-1}$]
$sgmscan^{inc}$ ⊙ $e_⊙$ [...,1, 0, ...,0, 1, ...]
  [...,$a_1^k$,$a_2^k$,...,$a_n^k$, $a_1^{k+1}$,...] =
  [...,$a_1^k$,$a_1^k$⊙$a_2^k$,...,$a_1^k$⊙...⊙$a_n^k$, $a_1^{k+1}$,...]
scatter [$a_0$, $a_1$, $a_2$, $a_3$, ...,$a_{n-1}$]
  [2, −1, 0, 3] [$b_0$, $b_1$, $b_2$, $b_3$] =
  [$b_2$, $a_1$, $b_0$, $b_3$, ...,$a_{n-1}$]
```

Figure 1: Futhark data-parallel array combinators.

to be explicit in the program, Futhark can be seen as a *sequentialising* compiler that attempts to efficiently sequentialise the parallelism in excess of what the hardware can support, thus enabling opportunities for locality-of-reference optimisations.

In this sense, the compiler supports a code transformation, named *moderate flattening* [33], that translates arbitrarily-nested, but *regular*[4] parallelism to a flat form that can be straightforwardly and efficiently mapped to the underlying GPU hardware, in the common case. (Whereas perfectly-nested parallel constructs can easily be translated to flat parallelism, it is not straightforward to do so with imperfectly-nested constructs.)

Futhark also implements a notion of *incremental flattening* [34], which generates multiple code versions in situations where the optimal optimisation strategy is sensitive to the input dataset.[5] The dynamic code selection is auto-tuned globally, resulting in one program that offers (in most cases) quasi-optimal performance for all datasets. However, given a problem exhibiting irregular parallelism, in general, to achieve a work efficient algorithm,[6] the Futhark programmer is required to implement the flattening strategy by hand, which is not necessarily straightforward [22] unless the problem is particularly simple [23]. The programmer may choose to apply a non-work-efficient padding approach instead (i.e., treat all problems to be of the same size), which might be the most efficient strategy in practice.

Finally, Futhark supports integration with various mainstream programming environments, such as Python [29].

3.1 Data-Parallel Functional Notation

We use (i) `[n]`α to denote the type of an array whose `n` elements have type α, (ii) `[`a_1`,...,`a_n`]` to denote an array literal, and (iii) `(a,b)` to denote a tuple (record) value. Applying a function `f` on two arguments `a` and `b` is written `f a b`.

The semantics of several key parallel operators are presented in Figure 1: `iota` applied to an integer `n` creates the array containing elements from `0` to `n-1`, and `replicate n v` creates an array of length `n` whose elements are all `v`. `map` produces a result array by applying its function argument `f` to each element of its input array. The function can be declared in the program or can be an anonymous function; for example, `map (`λ`x->x+1) arr` adds one to each element of `arr`. Similarly, `map2` applies its function argument to (corresponding) elements from the array arguments. `reduce` successively applies a binary-*associative* operator $\oplus$ to all elements of its input array ($e_\oplus$ denotes the neutral element).

`scan` [5] is similar to `reduce`, except that it produces an array of length `n` containing all prefix sums of its input array: the inclusive scan ($\texttt{scan}^{inc}$) starts from the first element of the array, and the exclusive scan ($\texttt{scan}^{exc}$) starts from the neutral element. Segmented scan (`sgmscan`) has the semantics of a `scan` applied on each subarray of an irregular array of subarrays. The latter is represented (i) by a flag array made of zeroes and ones in which a one denotes the start of a (new) subarray, and (ii) by a matching-length flat array containing all elements of all subarrays. For example, flag `[1,0,1,0,0,0,1]` denotes an array with three rows, having two, four and one elements, respectively. Segmented scan can be implemented as a `scan` with a modified operator.

The last operator, `scatter x is vs` updates in place the array `x` at indices contained in array `is` with the values contained in array `vs`, except that out-of-bounds indices are ignored (not updated). For example, in Figure 1, value b_1 was not written in the result because its index `-1` is out of bounds.

Finally, the notation supports the usual unary/binary operators and (normalised) `let` bindings, which have the form `let a=`e_1 `let b=`e_2`...in` e_n and are similar to a block of statements followed by a return denoted by keyword `in`. In-place updates to array elements are allowed and are written as `let arr[i]=x`. The notation supports `if` expressions, of form `if c then e2 else e3` and semantics similar to the C ternary operator `c? e1 : e2`, and loop expressions of the form: `loop (x) for i<n do e`. Here, `x` is a loop-variant variable that is initialised for the first iteration with an in-scope variable bearing the same name. `loop` executes iterations `i` from `0` to `n-1`, and the result of the loop-body expression `e` provides the value of `x` for the next iteration.

3.2 An Implementation of the Naive Algorithm

We first present a Futhark implementation of the naive LSMC algorithm, as specified in the original paper. The Futhark function lsmc_naive, which implements the main part of the algorithm is listed in Figure 2. The function takes as arguments (1) a two-dimensional array containing generated paths, (2) the maturity time (T) as a year fraction, (3) a risk-free interest rate r used for discounting, and (4) a payoff function pFun.

4 DESIGN AND IMPLEMENTATION

First of all, we emphasise that, in our implementation of LSMC, we follow the same algorithmic choices as taken by NVIDIA in their CUDA implementation [19]. However, as we could not find any published material on the exact linear algebra transformations that are undertaken, we specify the process in detail in the following sections. Therefore, we present the algorithmic consideration and an efficient approach to an implementation of a financial algorithm

[4] By *regular* parallelism we mean that the size of the inner-parallel operators is invariant to the outer-parallel nest.

[5] For example, the optimal GPU code for dense matrix-multiplication depends on the sizes of the matrix dimensions: If enough parallelism is available in the outer two parallel levels, then the dot-product dimension should be sequentialised, thus enabling various tiling strategies that optimise temporal locality. Otherwise, the inner dimension should also be executed in parallel.

[6] A parallel algorithm is *work efficient* if the work (i.e., the number of operations) performed by the algorithm is of the same asymptotic complexity as the work performed by the best known sequential algorithm that solves the same problem.

```
let lsmc_naive [paths] [steps]
    (Ss: [paths][steps]real)
    (T: real) (r: real) (pFun: real → real)
  : real =
  let Sst = transpose Ss
  let dt = T / (real (steps-1))
  -- compute discount factors
  let disc = map (λi → exp(r*real(-(i+1))*dt))
                 (iota (steps-1))
  -- prepare initial payoffs
  let Ps = map (λji →
                let j = ji / steps
                let i = ji % steps
                in if i < steps-1 then zero
                   else pFun(Ss[j,i])
                ) (iota (paths*steps))
  -- iteratively update the payoffs going backwards
  let (Ps, _) =
    loop (Ps, h) = (Ps, steps - 1)
    while h >= 1 do
      -- compute paths that are in-the-money
      let pickedpaths =
         filter (λj → pFun(Sst[h,j]) > zero)
                (iota paths)
      -- prepare for and perform regression
      let Y = map (λj →
        map (λi → disc[i]*Ps[j*steps + h+i])
            (iota (steps-h))
        |> reduce (+) zero) pickedpaths
      let Xt = map (λi →
                    map (λj → Sst[h,j] ** (real i)
                        ) pickedpaths
                   ) (iota 3)
      let X = transpose Xt
      let beta = Mat.matvecmul_row
          (Mat.matmul Xt X |> Mat.inv)
          (Mat.matvecmul_row Xt Y)
      let exVals =
          map (λj → pFun(Sst[h,j])) pickedpaths
      let contVals = map (λj →
          let sst = Sst[h,j]
          in (.0) <|
             loop (racc,sacc) = (0.0,1.0) for k < 3 do
               (racc + sacc * regY[k],
                sacc * sst)
        ) pickedpaths
      let (updInds, updVals) = unzip <|
        map (λji →
          let j = ji / steps
          let i = ji % steps
          let j' = pickedpaths[j]
          in if (contVals[j] < exVals[j]) then
                if (i != h) then (j' * steps + i, zero)
                else (j' * steps + i, exVals[j])
             else (-1, zero)
        ) (iota ((length pickedpaths)*steps))
      let Ps = scatter Ps updInds updVals
      in (Ps, h - 1)
  -- compute the discounted mean
  let prices =
    map (λj →
          map (λi → disc[i]* Ps[j*steps + i+1])
              (iota (steps-1))
          |> reduce (+) zero
        ) (iota paths)
  in Stats.mean prices
```

Figure 2: Futhark code for the naive LSMC algorithm.

for a widespread case of Monte Carlo simulation for American Option Pricing, which is frequently reimplemented across financial institutions. This is, to our knowledge, the first state-of-the-art high-level approach to this particular problem available to the public.

4.1 Main Considerations

The main inefficiency of the naive (straightforward) implementation of the LSMC algorithm is that $A^\dagger = (A^T A)^{-1} A^T$ is fully computed at each time step. We optimise this by means of an algorithmic refinement which aims to separate the part of the computation of $A^\dagger$ that is intrinsically dependent within the time-step loop from the one that is not. The latter, independent part can thus be precomputed in parallel before the time-step loop is entered. The algorithmic change consists, at a very high level, of working with a QR decomposition of $A = QR$, where the R matrices have small dimensionality (3×3 in our case) and can be efficiently precomputed in parallel for all time steps. With this, the computation inside the time-step loop is reduced to $A^\dagger = R^{-1} Q^T$.

Furthermore, the Q^T matrix is not actually manifested in memory, but rather computed on the fly from the sample matrix and fused in the multiplication with R^{-1}. This requires some redundant computation, but significantly decreases the number of accesses to global memory, which are orders of magnitude slower than scalar arithmetic. Finally, the sample matrix is computed in transposed layout in order to optimise spatial locality (i.e., coalesced accesses to global memory on GPU).

4.2 Building the Pseudo-Inverse Efficiently

We take our design goals into consideration and change the algorithm to adhere to parallel computation on GPUs. This approach was first adapted in the original CUDA implementation by NVIDIA that we are trying to match in performance [19, 39].

For the sake of brevity, we assume that we work with one system of equations $Ax = b$, although one per each time step i needs to be solved. We follow the standard practice by applying Singular Value Decomposition (SVD) $A = U\Sigma V^T$ to reduce the dimensions of A and build the Moore-Penrose pseudo-inverse of a tall matrix $A^\dagger = V\Sigma^{-1}U^T$. Next, $A^\dagger$ is used to compute the solution $\hat{x} = V\Sigma^{-1}U^T b$.

Yet, our goal is to build $A^\dagger$ efficiently. This is due to the fact that we need to construct one matrix A for each time step. Thereby it is simply too expensive in execution time and memory required to naively compute SVD of A. To remedy this problem, we start with a QR decomposition of $A = QR$, and use the fact that R is much smaller than A. We specialise the algorithm to work with a 3-degree parameterisation in terms of a quadratic polynomial. Therefore, R is of size 3×3 in our case. We compute SVD of $R = U_R \Sigma_R V_R^T$ to build the SVD of $A = QU_R\Sigma_R V_R^T$.

The QR factorisation is typically performed using householder transformation. However, the naive application of it would mean $3 \times n \times n$ memory access as that many elements would need to be updated. The efficient solution comes from the observation that, in our case, matrix R can be constructed using only 8 scalars:

$$S_0, S_1, S_2, \sum_{i=0}^{itm-1} S_i^0, \sum_{i=0}^{itm-1} S_i^1, \sum_{i=0}^{itm-1} S_i^2, \sum_{i=0}^{itm-1} S_i^3, \sum_{i=0}^{itm-1} S_i^4,$$

```
map(n)                  -- Path Generation
  loop(m)
transpose
map(m)                  -- SVD Preparation
  loop(n)
    scan(chunk)
  map(n) |> reduce(n)
  map(n)
map(m)
loop(m)                 -- Main Regression Loop
  map(n)
  map(n)
reduce(n)
```

Figure 3: The high-level view of the implemented optimised algorithm structure presented as a combination of parallel constructs. It consists of 3 parts with n denoting the number of paths and m denoting the number of time steps. The `transpose` function performs matrix transposition.

where S_i is the asset spot price on the ITM path i. Three first scalars are asset spot prices for the first ITM paths found when traversing the paths from path 0. The first sum can be translated to a total number of ITM paths found. The remaining four sums are consecutive powers $(1, \ldots, 4)$ of spot prices associated with each found ITM path. These scalars are prepared as part of SVD preparation. We implement a custom function that uses these scalars to build the matrix R. Furthermore, thanks to QR factorisation we get a simple formula for the pseudo-inverse. We use the fact that A is left-invertible, so its columns are linearly independent. We have

$$A^T A = (QR)^T (QR) = R^T Q^T QR = R^T R,$$

so

$$A^\dagger = (A^T A)^{-1} A^T = (R^T R)^{-1} (QR)^T = R^{-1} R^{-T} R^T Q^T = R^{-1} Q^T.$$

The final equation that we solve in the main loop for each time step is $\hat{x} = R^{-1} Q^T b$. The R^{-1} is precomputed for each time step before the loop using SVD like this: $R^{-1} = V_R \Sigma_R^{-1} U_R^T$. The orthogonal matrix $Q^T = R^{-T} A^T$ does not need to be stored for each time step and can instead be computed on-the-fly. We again use the fact that (1) R^{-1} is by now already precomputed using SVD and (2) matrix A can itself be computed on-the-fly, where, for our case, each row comprises 3 elements: $1, S_{ij}, S_{ij}^2$, i.e., it can be computed from vector S_i that consists of asset spot prices for ITM paths for a given time step i. Naturally, they need to be processed in transformed form.

4.3 Optimised Algorithm

The optimised version of the algorithm follows closely the implementation proposed by NVIDIA [19, 39]. We focus on the goal to match the performance of this public benchmark implementation. The obtained algorithm outlined in Figure 3 is implemented using a nested composition of sequential **loop**s and parallel **map**, **reduce**, and **scan** constructs. The code in Figure 4 demonstrates the main lsmc_opt function of the optimised algorithm. The function takes the following arguments: (1) number of time steps m, (2) number of paths n, (3) a function to verify if the option is ITM is_itm, (4) a payoff function payoff, (5) the time step size dt as a fraction of year, (6) initial asset spot price at the current day S0, (7) a risk-free

```
let lsmc_opt (m: i32) (n: i32)
    (is_itm: real → i32)
    (payoff: real → real) (dt: real)
    (S0: real) (r: real) (sigma: real) (seed: i32)
    (min_itm: i32) (CHUNK: i32) : real =
  -- Path Generation
  let paths = generate_samples_and_paths
      prng_seed m n S0 dt r sigma payoff
  -- SVD Preparation
  let Sst = transpose paths
  let (svds, all_otms) = prepare_svds Sst is_itm
      min_itm CHUNK
  -- Main Regression Loop
  let expmrdt = exp(-r*dt)
  let (cashflows, _) =
    loop (cashflows, i) = (Sst[m - 1], m-2)
    while i >= 0 do
      let betas = compute_betas is_itm svds[i] Sst[i]
          cashflows all_otms[i]
      let new_cashflows = update_cashflows payoff
          expmrdt betas Sst[i] all_otms[i] cashflows
    in (new_cashflows, i - 1)
  in expmrdt * (reduce (+) zero cashflows) / n
```

Figure 4: Futhark code for the main regression loop.

interest rate r used for discounting, (8) volatility sigma, (9) seed for Random Number Generator and 2 helper parameters that determine the amount of computation that should be performed sequentially. The function returns the calculated option price.

In the next sections, we give a detailed description of the implementation and optimisations involved in the algorithm. The three main parts of the algorithm are: path generation in Section 4.3.1, SVD preparation in Section 4.3.2, and main regression loop in Section 4.3.3. Each section is accompanied with a code listing presenting a Futhark implementation of a function for the given part.

4.3.1 Path Generation. The computational effort of a Monte Carlo simulation is determined by the number of 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 [25]. In American option pricing case, the number of time steps m is bound to the number of early-exercise opportunities and is usually much smaller than n. Path generation part consists of two sub-parts: random number generation and path generation.

For the first part, we use a minimum standard pseudo-random number generator (RNG) in a parallel skip-ahead fashion. After seeding the RNG, we draw $m \times n$ random samples by splitting the RNG into n sub-RNGs for each path and than sequentially drawing a sample for each time step of the given path. The samples are independent from each other. For the process, that we want to simulate, we need samples drawn from Gaussian (standard normal) probability distribution. We achieve it in two steps. First, we draw samples from a uniform probability distribution using the RNG. Afterwards, we use the *Inverse Normal Cumulative Density Function* (CDF) to produce normally-distributed samples out of the generated uniforms. As there exists no exact formula for Inverse Normal CDF, we need to use an approximation algorithm. We implement the *Beasley-Springer-Moro* algorithm known for its speed and accuracy following the procedure described in [25]. For the simulation, every sample is turned into an asset spot price instance at every simulation time step (or early-exercise opportunity). We use the standard *Geometric Brownian Motion GBM*(r, σ^2) with a mean (drift) equal to the risk-free interest rate r and variance (diffusion) equal to σ^2 (square of volatility). We use the generated normally-distributed samples to simulate a stochastic process. In practice, as the process is a Markov chain, we know that the current step is independent of the past realisations of the process.

In our case, we deal with only one stochastic factor—the underlying asset spot price, as the American option, that we price, depends on only one underlying variable. All paths are therefore independent, which allows us to parallelise the generation efficiently across paths by having one thread generate one whole path. The assumption of having one underlying is not essential for parallelism, however. Many stochastic processes could be generated in parallel as long as we would adjust the simulation for the correlations between the stochastic variables, which is a necessary step in practice. The code in Figure 5 presents a compact Futhark implementation of path generation. For each path and step, the UniDist.rand function draws first a single random number from a uniform distribution. The function compGbmNormalStep transforms it to a standard normal distribution and computes a current GBM step. It uses function NormRealDist.invNormalCdf to approximate the Inverse Normal CDF of a uniform sample. At the last time step the cash flow is known (given the payoff function), as this is the last time the option can be exercised.

Performance Enabler. We gain most performance here by fusing the random sample generation and path generation together into one step like in lines 19–24 in the code in Figure 5. We perform these actions for each step based on the observation that we work on the same array for both actions as well as each sample is independent from the all other ones. This way we read from and write to the device global memory only once and thereby save the redundant intermediate memory accesses, that are costly to execute compared to compute instructions.

```
1  let compGbmNormalStep (drift: real) (vol: real)
2      (x: real) : real =
3    drift * exp(vol * (NormRealDist.invNormalCdf x))
4
5  let generate_samples_and_paths (seed: i32)
6      (m: i32) (n: i32) (S0: real) (dt: real)
7      (r: real) (sigma: real) (payoff: real → real)
8    : [n][m]real =
9    let rng = minstd_rand.rng_from_seed [seed]
10   let std_dist = (0.0, 1.0)
11   let rngs = minstd_rand.split_rng n rng
12   let drift = exp((r-0.5*sigma*sigma)*dt)
13   let dtSigma = sigma * mysqrt(dt)
14   in map (λr →
15     let path = replicate m 0.0
16     let (path', _, _) =
17       loop (path, rng, acc) = (path, r, 1.0)
18       for i < m do
19         let (rng, num) = UniDist.rand std_dist rng
20         let W = compGbmNormalStep drift dtSigma num
21         let acc' = acc * W
22         let v = acc' * S0
23         let v' = if i < m - 1 then v else payoff v
24         let path[i] = v'
25         in (path, rng, acc')
26     in path'
27   ) rngs
```

Figure 5: Futhark code for path generation.

4.3.2 SVD Preparation. This part is run before entering the main regression loop and covers the main algorithmic optimisation. The main advantage of this approach is that SVD for R in each time step can be processed in parallel. As R is small, the intermediate variables easily fit into registers of one streaming multiprocessor (SM). The number of SVD matrices to prepare depends on the number of time steps m. This part is compute-intensive, but at the same time the parallelism is limited by the fact that m is usually much smaller than n. A sequential loop is needed to find the first three ITM paths to get the asset spot prices to build R matrix. In the body of prepare_svds function, the eight required scalars are gathered and computed. They are subsequently passed to svd_3x3 function that performs the QR decomposition and SVD decomposition for R and R^{-1}. It start with assembling the R matrix from the 8 scalars and afterwards uses the iterative Jacobi method to determine the inverse matrix R^{-1}. The function returns 6 upper elements of matrix R and 6 upper elements of the inverse matrix R^{-1}, as the matrices are orthogonal.

Performance Enabler. The key to performance is that we not only perform computation in parallel on the outer level across all time steps m, but that we also enable inner parallelism in the computation for each time step.

First of all, SVD preparation benefits significantly from the intra-group parallelism in the first map (lines 4–29 in the code in Figure 6), which finds the first three ITM paths. The spot prices on these paths are needed for computing the matrix R. The computation works on CHUNK paths in parallel. It is achieved by a combination of parallel constructs like **map**s, **reduce**s, an exclusive scan scanExc, and **scatter**. The CHUNK parameter depends the level of intra parallelism. The smaller its value, the faster this part performs. However, the value cannot be smaller than the number of ITM paths needed for constructing the R SVD matrices. This number is determined by the min_itm parameter, which is fixed to 4, one larger than the dimension size of R.

The next beneficial optimisation is the application of segmented reduction in lines 31–44, which enables parallelism in gathering the remaining 5 scalars. Afterwards, the **map3** in line 45 works in parallel across time steps, but internally, for each time step, it uses the matching scalars to compute the partial data in a sequential manner in the call to svd_3x3.

4.3.3 Main Regression Loop. This part is where the least squares system of equations is regressed, the continuation value is computed and the cash flow per each time step is updated. It can be seen in the code in Figure 4 in lines 15–22. This loop has $m - 1$ iterations. The computation in the loop is greatly simplified thanks to the SVD preparation that is performed before entering the loop as most of the sequential computation was performed there. The code in Figure 7 shows the implementation. Function compute_betas

```
let prepare_svds [m] [n] (Sst: [m][n]real)
        (is_itm: real → i32) (min_itm: i32)
        (CHUNK: i32) : ([m][12]real, [m]i32) =
  let svds = map (λSs →
    let svds = replicate 12 zero
    -- Loop to find 3 first ITM paths
    let (svds, _, _, _) =
      loop (svds, found_paths, path_offs, exit)
          = (svds, 0, 0, false)
      while (!exit && found_paths < 3
          && path_offs < n) do
        let Ss_chunked = map (λi →
            if i + path_offs < n
            then Ss[i+path_offs]
            else zero) (iota CHUNK)
        let itms = map is_itm Ss_chunked
        let exit = reduce (&&) true <| map (==0i32)
          itms
       let scn_ms = scanExc (+) 0 itms
        let tot_sum = scn_ms[CHUNK-1] + itms[CHUNK-1]
        let inds = map2 (λin_m sm →
            if in_m == 1i32 && found_paths+sm < 3
            then found_paths+sm
            else -1) itms scn_ms
        let svds = scatter svds inds Ss_chunked
        let found_paths = found_paths + tot_sum
        in (svds, found_paths, path_offs+CHUNK, exit)
    in svds
    ) Sst
  let (ms, sums, all_otms) = unzip3 <|
    map (λSs →
      let itms = map is_itm Ss
      let ms = reduce_comm (+) 0i32 itms
      let sums = map2 (λin_m S →
        if in_m == 1i32
        then (S, S*S, S*S*S, S*S*S*S)
        else (zero, zero, zero, zero)
      ) itms Ss
      |> reduce_comm tuple4_sum_op
          (zero, zero, zero, zero)
      let all_otm = if (ms < min_itm) then 1i32
                    else 0i32
      in (ms, sums, all_otm)
    ) Sst
  let svds = map3 svd_3x3 ms sums svds
  in (svds, all_otms)
```

Figure 6: Futhark code for SVD preparation.

computes β coefficients required for regression through a multiplication of pseudo-inverse $A^{\dagger}$ and cash flow vector. Afterwards, the update_cashflows estimates a payoff based on βs for each path and determines continuation value for each of them, comparing the estimated payoff with a current payoff of the option.

Performance Enabler. Thanks to the SVD preparation, before start of the main regression loop, the computational work in each loop iteration is significantly reduced. The other reason is the reduced size of the matrices that are being processed. All the remaining computation is performed in parallel across n paths. As presented in code in Figure 7, compute_betas uses a `map` (line 8) followed by a `reduce` (line 16). The update_cashflows follows with one more `map` in line 23. The performance of the loop is highly dependent on the

```
 1 let compute_betas [n] (is_itm: real → i32)
 2     (svds: [SLOTS]real) (Ss: [n]real)
 3     (cashflows: [n]real) : [](real, real, real) =
 4   let R00 = svds[0]
 5   -- Initialise R and W matrices from svds
 6   -- and compute inverse of R and W
 7   -- ...
 8   in map2 (λS i →
 9     -- Compute Qis. The elements of the Q matrix
10     -- in the QR decomposition.
11     -- ...
12     let cashflow = if (is_itm S) == 1i32
13       then cashflows[i] else zero
14     in (WI0*cashflow, WI1*cashflow, WI2*cashflow)
15   ) Ss (iota n)
16     |> reduce_comm tuple3_sum_op (zero, zero, zero)
17
18 let update_cashflows [n]
19     (payoff: real → real) (expmrdt: real)
20     (beta: [](real,real,real))
21     (Ss: [n]real) (cashflows: [n]real)
22   : [n]real =
23   map2 (λS path →
24     let old_cashflow = expmrdt * cashflows[path]
25     let cur_payoff = payoff S
26     let (beta0, beta1, beta2) = beta
27     let estimated_payoff =
28       (beta0 + beta1 * S + beta2 * S * S) * expmrdt
29     in if cur_payoff <= estimated_payoff
30        then old_cashflow else cur_payoff
31   ) Ss (iota n)
```

Figure 7: Futhark code for the main regression loop. It consists of a computation of βs for assessing the continuation value with a subsequent update of the cash flows.

number of time steps, as they are processed sequentially, because of the data dependency between consecutive time steps.

4.4 Motivation for the Functional Approach

As a final remark to the presented Futhark code, we motivate some virtues of the functional programming approach that can be especially valuable for non-technical domain experts. Anyone, who, on a regular basis, programs in parallel low-level languages targeting multi-core architectures, will make the immediate observation that the code contains no explicit kernel setup or device memory management. In fact, the code is agnostic to the underlying platform and most of the performance optimisations are provided by the aggressive optimisation and parallelisation strategies implemented by the compiler. The effect is that the same Futhark code base is portable across architectures. The program can be compiled to run sequentially on a single CPU core or be compiled to run on (and utilise fully) a massively-parallel GPU. Moreover, when writing Futhark code, the programmer can focus on expressing the algorithmic details using Futhark's Second-Order Array Combinators (SOACs) that mimic the higher-order functions found in conventional functional languages. In Futhark, SOACs have sequential semantics, but permit parallel execution. The purely functional nature of Futhark allows the compiler to apply high-level optimisations. In terms of performance portability, Futhark supports nested parallelism, which means that the code can be further auto-tuned to support efficiently all the available parallelism exposed by the hardware, which is a rather cumbersome, and often impossible, task to perform manually.

5 EXPERIMENTAL RESULTS

In this section, we present different experimental tests and discuss their results. We validate the simulation accuracy and measure the performance of the implementation by comparing it to other established benchmarks. We run the 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 Volta *FP64* cores, 16 GB HBM2) using CUDA 10.1.

5.1 Accuracy

To start with, we compare the pricing results with an established benchmark to validate correctness of our implementation. Table 1 presents a comparison of different implementations of American Option Pricing. We are pricing an American put option with a fixed strike and constant risk-free rate. The remaining parameters vary as specified in the original paper by Longstaff-Schwartz [37]. Columns FDM and LSMC stand for the results from the original paper. The two remaining columns comprise results for CUDA implementation, that we used as a benchmark algorithm, and our Futhark implementation. That said, we mention that the LSMC simulation from the original paper uses antithetic sampling, which cuts the number of random samples in half. This algorithmic optimisation leads to a reduced variance of the sample paths as well as an improvement in the overall accuracy of the simulation.

The simulation results compared to FDM method have a low error. The difference between simulation results varies slightly for different sets of parameters, but, in general, they are insignificant and are the outcome of using different RNGs. The same is valid for both the original CUDA and Futhark implementations.

5.2 Performance Test Case

The pricing test case is presented in Table 2. It is an example of a typical put option that is ITM on the calculation day.

The performance results are presented in Table 3. The correctness is validated against the benchmark binomial tree numerical method for the same problem. We want to obtain a value that is as close as possible to this benchmark. Futhark targets different hardware with different backends. We test the CUDA (**V1**) and OpenCL (**V2**) backends, and observe identical execution times for both versions. In terms of the speedups, **Ref** is only 1.11× faster than **V2**, what can be considered insignificant. Furthermore, we do not observe large discrepancies in terms of partial execution times among the three main parts of the algorithm. This fact demonstrates that Futhark auto-generated low-level code is similar in complexity and on par in performance with one that is hand-tuned. Some parts run faster, while other would benefit from further optimisation (e.g., changing the algorithm). The (17%) overhead in the **Path** part is due to the internal implementation of the RNG. We have used a different RNG in comparison to **Ref**, which uses CURAND_RNG_PSEUDO_MRG32K3, a member of the Combined Multiple Recursive family of pseudo-random number generators. The 13% difference in **Main** is caused by a more optimal handling of memory copies compared to **Ref**. The effect is amplified, because the loop has 99 iterations.

Table 1: Comparison between results from the original paper (finite difference method (FDM) and LSMC) and LSMC implementations in CUDA and Futhark. The strike price of the put option is 40 and the risk-free rate is 0.06. The remaining parameters are as indicated. All LSMC simulations are done with 100.000 paths and 50 time steps per year.

S_0	σ	T	FDM	LSMC	CUDA	Futhark
36	0.20	1	4.478	4.472	4.460	4.465
36	0.20	2	4.840	4.821	4.821	4.826
36	0.40	1	7.101	7.091	7.077	7.092
36	0.40	2	8.508	8.488	8.514	8.518
38	0.20	1	3.250	3.244	3.232	3.239
38	0.20	2	3.745	3.735	3.736	3.739
38	0.40	1	6.148	6.139	6.131	6.147
38	0.40	2	7.670	7.669	7.670	7.661
40	0.20	1	2.314	2.313	2.307	2.313
40	0.20	2	2.885	2.879	2.873	2.878
40	0.40	1	5.312	5.308	5.290	5.319
40	0.40	2	6.920	6.921	6.914	6.909
42	0.20	1	1.617	1.617	1.613	1.612
42	0.20	2	2.212	2.206	2.205	2.205
42	0.40	1	4.582	4.588	4.578	4.590
42	0.40	2	6.248	6.243	6.231	6.234
44	0.20	1	1.110	1.118	1.104	1.104
44	0.20	2	1.690	1.675	1.682	1.680
44	0.40	1	3.948	3.957	3.945	3.952
44	0.40	2	5.647	5.622	5.628	5.637

Table 2: Model and simulation parameters for the American option pricing. The reference value is obtained with a different (binomial tree) numerical method.

Model Parameters	Value
Option Type, Payoff	Put, $\max(K - S)$
Initial Spot price (S_0)	80.0
Strike price (K)	90.0
Time to maturity (T)	1 year
Risk free rate (r)	5%
Volatility (σ)	30%
Simulation Parameters	
Time steps/Early Exercise dates	100
Paths	1.024.000
Reference value (Binomial Tree)	13.804

5.3 Performance Scalability

5.3.1 Fixed number of paths, various number of time steps. As the next step we test the scalability behaviour of Futhark implementation **V2** by gradually changing the number of time steps or paths. These are the main parameters that determine the size of the computation involved in a Monte Carlo simulation. Consequently, we reuse the test case from Table 2 and compare against the benchmark **Ref** for different combinations of these two simulation parameters.

Table 3: Execution times for the test case. Ref is the original 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. The execution times are given in *ms* and averaged based on 250 runs. *Path* stands for Path Generation part, *SVD* — SVD Preparation, and *Main* for the Main Regression Loop. The Δ column compares the speedups against the slowest execution time. Obtained values are presented in the *Val* column.

	Path	SVD	Main	Total	Δ	Val
Ref	4.7 (30%)	1.8 (12%)	8.9 (58%)	**15.4**	1.11×	13.778
V1	8.0 (47%)	1.4 (8%)	7.7 (45%)	**17.1**	1.00×	13.789
V2	8.0 (47%)	1.4 (8%)	7.7 (45%)	**17.1**	1.00×	13.789

First of all, we fix the number of paths to a relatively high number 1.024.000 and test against 5 different numbers of time steps. High number of paths allows for massive parallelism across the path dimension and thus full utilisation of the GPU hardware. Figure 8 shows the results of this experiment. Contributions of three algorithmic parts are distinguished and sum up to a total execution time for each tested case.

The main observation is that for the low number of time steps, Futhark **V2** is faster than benchmark CUDA **Ref**. In particular, for very few time steps like 10, it is ~ 2.5× faster. The difference diminishes with increasing number of time steps to match at ~ 100 time steps. For more time steps, **Ref** is slightly faster than **V2**, but the ratio is maintained with increasing time steps. For instance, it is 1.25× faster for 250 time steps.

The next observation is that, for many time steps, the **Path** part becomes the main computational bottleneck. It takes 40% for 10 time steps, but more than 52% for 250. This is caused by the necessary transposition of the layout that the sampled paths matrix is organised in. The transposition enables coalesced accesses to global memory on GPU across time steps, which benefits the performance of the **SVD** part. The next observation is that the contribution of **SVD** part decreases with more time steps. It is more efficiently implemented in **V2** as it is always faster than the one in **Ref**. Finally, we achieve better performance on the sequential part **Main** by ensuring that no device-to-host memory transfers are initiated. On GPU architectures such transfers introduce significant delays in execution, so all intermediate data should be kept on the device. The compute_betas and update_cashflows functions in 7 exchange an array of βs that is computed sequentially and not in parallel (it holds only three elements).

5.3.2 Fixed number of steps, various number of paths. We observe similar scalability, when the number of paths is increased. Futhark **V2** is faster than CUDA **Ref** on low number of paths. For 10240 paths, it is ~ 1.25× faster. The difference diminishes with increasing number of time steps to match at ~ 1024000 paths.

5.3.3 Various number of time steps and paths. Figure 10 presents the results for different combinations of time steps and paths. The ratio between them is kept so, that the required work as well as memory requirements are constant. These cases saturate the memory available on **V100**. We can see that the impact of the time steps on the overall execution time is slightly higher than the number of paths. For **V2** it goes from 39 ms to 45 ms. This is caused by the computations in **Main** loop, that need to be run one step at a time. In general, the performance of both **Ref** and **V2** is stable across different configurations, which shows that Futhark matches the CUDA performance. We can observe that Futhark **V2** is slightly (1 ms to 13 ms) slower than **Ref** on all cases.

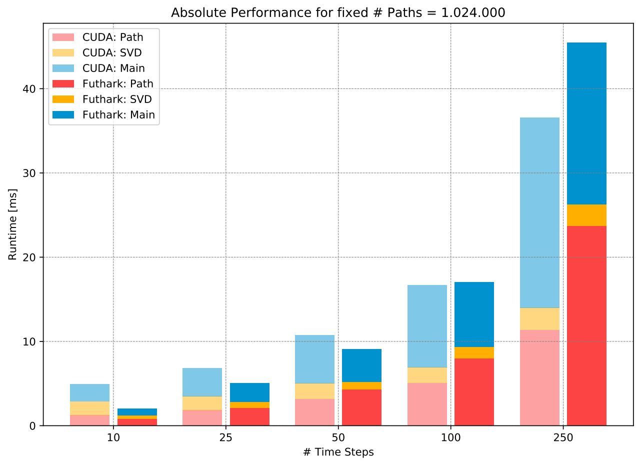

Figure 8: Execution time comparison of CUDA (Ref) and Futhark (V2). Absolute performance is presented for a fixed number of 1.024.000 paths. Execution time is given in *ms*.

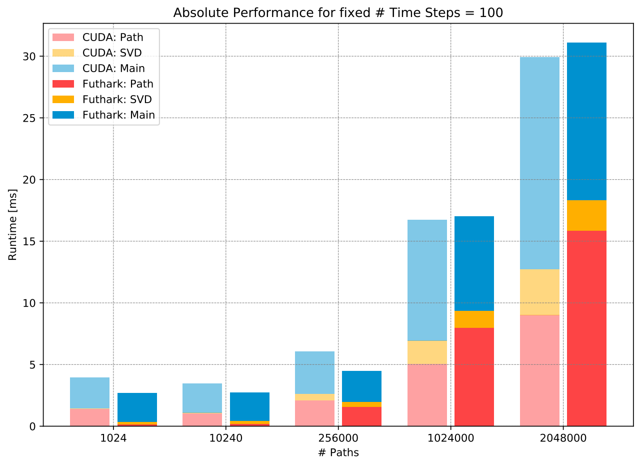

Figure 9: Execution time comparison of CUDA (Ref) and Futhark (V2). Absolute performance is presented for a fixed number of 100 steps. Execution time is given in *ms*.

6 RELATED WORK

The most efficient implementations of Monte Carlo based American option pricing are implemented in low-level dedicated data-parallel languages and frameworks, such as CUDA [1, 24, 43, 54]. Other efficient parallel implementations are based on task-parallel approaches [15, 16], which are suitable for multi-core architectures.

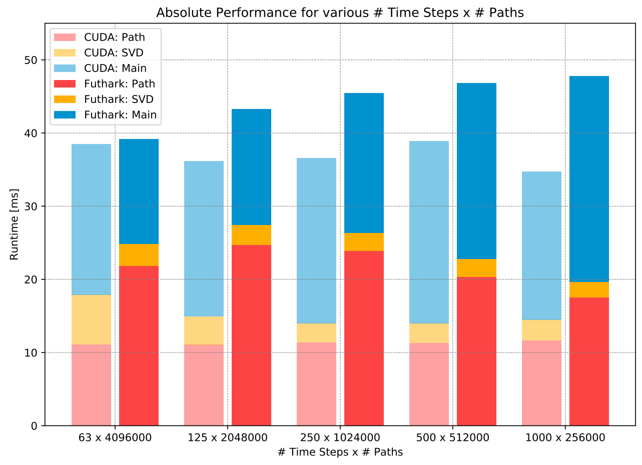

Figure 10: CUDA (Ref) and Futhark (V2) are compared in terms of execution times. Absolute performance is presented for different combinations of number of time steps and paths. Execution time is given in *ms*.

We are not aware of any accelerated implementations of American option pricing using functional languages.

Previous work has investigated the use of Futhark for implementing Monte Carlo based European option pricing [2, 40]. Whereas European option pricing is simpler than (and a special case of) American option pricing, the previous work covered a number of advanced features that the present work does not consider. In particular, the previous work on European option pricing considered European options with multiple underlying assets, Sobol sequence generation [30], and options that are so-called *path dependent*, meaning that the price of an option not only depends on the value of the underlying at maturity, but also on intermediate values of the underlying. Using the module language features of Futhark, we believe it will be possible to parameterise the implementation in such a way that multiple underlying assets and path dependence are features that are supported by the American pricing engine (we consider such advanced features future work). It is also straightforward to replace the pseudo-random number generation used with Sobol sequence generation.

There are quite a few functional language approaches aiming at generating efficient data-parallel GPU code for applications written using high-level array language constructs. Such high-level approaches include SaC [26, 27], a functional array-based language featuring a parallelisable with-loop construct, and Obsidian [17, 50, 51] and Accelerate [13], which are both domain specific languages embedded in Haskell. None of these approaches, however, feature arbitrary nested parallelism. Approaches that support arbitrary nested parallelism includes the seminal work on flattening of nested parallelism in NESL [6, 7], which was extended to operate on a richer set of values in Data-parallel Haskell [12], and the work on data-only flattening [55]. However, such general compiler-based flattening is challenging to implement efficiently in practice, particularly on GPUs [3]. Other promising attempts at compiling NESL to GPUs include Nessie [47], which is still under development, and CuNesl [55], which aims at mapping different levels of nested parallelism to different levels of parallelism on the GPU, but lacks critical optimisations such as fusion.

Imperative approaches typically rely on low-level analysis—of loop nests using affine indexing and branch condition—for discovering and optimising parallelism, for example the polyhedral model [8, 44]. Since the affine domain is too restrictive in practice, several techniques use explicit annotations to extend the applicability of polyhedral transformations [14, 46]. Other techniques rely on more dynamic analysis coupled with multi-versioned code generation, for example to identify sufficient conditions under which loops are parallel [38, 42] or to optimise locality of reference and communication [20, 41, 45, 49]. In this sense, our implementation uses Futhark's multi-version code generation facilities [34] to adapt the compilation to the particularities of datasets and hardware.

7 CONCLUSION

In this work, we present the results of the accelerated implementation of a well-known LSMC algorithm for a common financial use case of American Option Pricing. We choose to use a high-level functional approach to the implementation, express the algorithm using succinct parallel constructs and let the optimising compiler auto-generate an efficient parallel code that targets massively parallel hardware. For this purpose, we use the Futhark language and address GPUs as a suitable compute platform. We demonstrate that with this approach it is possible to achieve the execution times that are, in general, at the same level as the hand-tuned implementations in dedicated languages like CUDA, but there exist particular smaller cases, where the implementation beats the benchmark by up to 2.5×. This promising finding motivates further work on the optimisation compiler and the algorithm.

We consider the high-level functional specification as being much more suitable and accessible for the financial-domain experts than the low-level dedicated code, that is usually written by expert software developers. Its expressibility and modularity enables code maintainability, hiding the implementation details targeting particular parallel architecture, and instead turning focus to algorithmic and domain-specific consideration. It also facilitates algorithmic changes, so prevalent in the financial industry, with its multitude of financial instruments traded in the global markets.

REFERENCES

[1] L.A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. 2014. Pricing derivatives on graphics processing units using Monte Carlo simulation. *Concurrency and Computation: Practice and Experience* 26, 9 (June 2014), 1679–1697.
[2] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages.
[3] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the Gpu. In *Proc. of the 17th ACM Int. Conf. on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 247–258.
[4] Patrick Billingsley. 2012. *Probability and Measure.* John Wiley & Sons. Google-Books-ID: a3gavZbxyJcC.
[5] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Trans.* 38, 11 (1989), 1526–1538.
[6] Guy E Blelloch. 1990. *Vector models for data-parallel computing.* Vol. 75. MIT press Cambridge.
[7] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM (CACM)* 39, 3 (1996), 85–97.
[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of*

the 29th ACM Int. Conf. on Programming Language Design and Impl. (PLDI '08). ACM, New York, NY, USA, 101–113.

[9] Stephen Boyd and Lieven Vandenberghe. 2018. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares* (1st ed.). Cambridge Univ. Press.

[10] Christian Brugger, Javier Alejandro Varela, Norbert When, Songyin Tang, and Ralf Korn. 2015. Reverse longstaff-schwartz american option pricing on hybrid CPU/FPGA systems. In *2015 Design, Automation Test in Europe Conf. Exhibition (DATE)*. 1599–1602. ISSN: 1530-1591, 1558-1101.

[11] Jacques F. Carriere. 1996. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: Mathematics and Economics* 19, 1 (Dec. 1996), 19–30.

[12] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Int. Workshop on Decl. Aspects of Multicore Prog. (DAMP)*. 10–18.

[13] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Workshop on Decl. Aspects of Multicore Prog. (DAMP'11)*. ACM, 3–14.

[14] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral Optimizations of Explicitly Parallel Programs. In *Proc. of the 2015 Int. Conf. on Parallel Architecture and Compilation (PACT '15)*. IEEE, Washington, DC, USA, 213–226.

[15] Ching-Wen Chen, Kuan-Lin Huang, and Yuh-Dauh Lyuu. 2015. Accelerating the least-square Monte Carlo method with parallel computing. *The Journal of Supercomputing* 71, 9 (Sept. 2015), 3593–3608.

[16] A. R. Choudhury, A. King, S. Kumar, and Y. Sabharwal. 2008. Optimizations in financial engineering: The Least-Squares Monte Carlo method of Longstaff and Schwartz. In *2008 IEEE Int. Symp. on Parallel and Distributed Processing*. 1–11.

[17] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Workshop on Decl. Aspects of Multicore Prog. (DAMP'12)*. 21–30.

[18] Emmanuelle Clément, Damien Lamberton, and Philip Protter. 2002. An analysis of a least squares regression method for American option pricing. *Finance and Stochastics* 6, 4 (Oct. 2002), 449–471.

[19] Julien Demouth. 2014. Monte-Carlo Simulation of American Options with GPUs. http://on-demand.gputechconf.com/gtc/2014/presentations/S4784-monte-carlo-sim-american-options-gpus.pdf. At NVIDIA GPU Technology Conf.

[20] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Proc. of the ACM Int. Conf. on Prog. Lang. Design and Impl. (PLDI '99)*. ACM, New York, NY, USA, 229–241.

[21] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proc. of the ACM on Programming Languages* 2, ICFP, Article 97 (July 2018), 30 pages.

[22] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. 2018. *Parallel Programming in Futhark*. Department of Computer Science, University of Copenhagen. https://futhark-book.readthedocs.io

[23] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. 2019. Data-parallel Flattening by Expansion. In *Proc. of the 6th ACM Int. Workshop on Libraries, Languages and Compilers for Array Prog. (ARRAY '19)*. ACM, New York, NY, USA, 14–24.

[24] Massimiliano Fatica and Everett Phillips. 2013. Pricing American Options with Least Squares Monte Carlo on GPUs. In *Proc. of the 6th Workshop on High Perf. Comp. Finance (WHPCF '13)*. ACM, New York, NY, USA, 5:1–5:6.

[25] Paul Glasserman. 2004. *Monte Carlo methods in financial engineering*. Springer, New York.

[26] Clemens Grelck and Sven-Bodo Scholz. 2003. SaC - From High-Level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters* 13 (09 2003), 401–412.

[27] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU Programming Barrier with the Auto-parallelising SaC Compiler. In *Proc. of the Sixth Workshop on Decl. Aspects of Multicore Programming (DAMP '11)*. ACM, New York, NY, USA, 15–24.

[28] Espen Gaarder Haug. 2007. *The Complete Guide to Option Pricing Formulas* (2nd ed.). McGraw-Hill Education, New York.

[29] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Funct. High-Performance Comp. (FHPC'16)*. ACM, New York, NY, USA, 38–43.

[30] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2018. Modular Acceleration: Tricky Cases of Functional High-Performance Computing. In *Proc. of the 7th ACM Int. Workshop on Funct. High-Performance Comp. (FHPC '18)*. ACM, New York, NY, USA.

[31] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proc. of the 3rd ACM Int. Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 17–24.

[32] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proc. of the 2nd ACM Workshop on Functional high-performance computing*. ACM, 47–58.

[33] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proc. of the 38th ACM Int. Conf. on Prog. Lang. Design and Impl. (PLDI 2017)*. ACM, New York, NY, USA, 556–571.

[34] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proc. of the 24th Symp. on Principles and Practice of Parallel Prog. (PPoPP '19)*. ACM, New York, NY, USA, 53–67.

[35] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*.

[36] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proc. of the 6th ACM Int. Workshop on Funct. High-Performance Comp. (FHPC 2017)*. ACM, New York, NY, USA, 42–52.

[37] Francis A. Longstaff and Eduardo S. Schwartz. 2001. Valuing American Options by Simulation: A Simple Least-Squares Approach. *The Review of Financial Studies* 14, 1 (Jan. 2001), 113–147.

[38] Sungdo Moon and Mary W. Hall. 1999. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*. 84–95.

[39] NVIDIA. 2014. NVIDIA Developer Blog Code Samples repository at GitHub. https://github.com/NVIDIA-developer-blog/code-samples/tree/master/posts/american-options.

[40] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Proc. of the 1st ACM Workshop on Funct. High-performance Comp. (FHPC '12)*. ACM, New York, NY, USA, 61–72.

[41] Cosmin E. Oancea and Alan Mycroft. 2008. Set-Congruence Dynamic Analysis for Software Thread-Level Speculation (TLS). In *Procs. Langs. Comp. Parallel Comp.* 156–171.

[42] Cosmin E. Oancea and Lawrence Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Languages and Compilers for Parallel Comp.*, Sanjay Rajopadhye and Michelle Mills Strout (Eds.). Springer, 61–75.

[43] Gilles Pagès and Benedikt Wilbertz. 2012. GPGPUs in computational finance: massive parallel computing for American style options. *Concurrency and Computation: Practice and Experience* 24, 8 (2012), 837–848.

[44] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *Proc. of the 38th ACM Symp. on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 549–562.

[45] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2014. Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems. *ACM Trans. Parallel Comput.* 1, 1, Article 7 (Oct. 2014), 37 pages.

[46] Chandan Reddy, Michael Kruse, and Albert Cohen. 2016. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Proc. of the 2016 Int. Conf. on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 87–97.

[47] John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at Compilers for Parallel Comp. Workshop. Imperial College, London.

[48] Albert N. Shiryaev. 2016. *Probability-1* (3rd ed.). Springer-Verlag, New York.

[49] Michelle Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. of the IEEE* PP (08 2018), 1–15.

[50] Joel Svensson. 2011. *Obsidian: GPU Kernel Programming in Haskell*. Ph.D. Dissertation. Chalmers University of Technology.

[51] Joel Svensson, Mary Sheeran, and Koen Claessen. 2011. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *Proc. of the 20th Int. Conf. on Impl. and Appl. of Funct. Lang. (IFL'08)*. Springer-Verlag, Berlin, Heidelberg, 156–173.

[52] J. N. Tsitsiklis and B. Van Roy. 2001. Regression methods for pricing complex American-style options. *IEEE Trans. on Neural Networks* 12, 4 (July 2001), 694–703.

[53] Javier Alejandro Varela, Christian Brugger, Songyin Tang, Norbert Wehn, and Ralf Korn. 2015. Pricing High-Dimensional American Options on Hybrid CPU/FPGA Systems. In *FPGA Based Accelerators for Financial Applications*, Christian De Schryver (Ed.). Springer Int. Publishing, Cham, 143–166.

[54] Shuai Zhang, Zhao Wang, Ying Peng, Bertil Schmidt, and Weiguo Liu. 2017. Mapping of option pricing algorithms onto heterogeneous many-core architectures. *The Journal of Supercomputing* 73, 9 (Sept. 2017), 3715–3737.

[55] Yongpeng Zhang and Frank Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *Proc. of the 2012 41st Int. Conf. on Parallel Processing (ICPP'12)*. IEEE, Washington, DC, USA, 340–349.