A Financial Benchmark for GPGPU Compilation

Christian Andretta
Vivien Begot
Jost Berthold
Martin Elsman
Troels Henriksen
Maj-Britt Nordfang
Cosmin E. Oancea

Technical Report no. 2015/02
ISSN: 0107-8283
A Financial Benchmark for GPGPU Compilation

Christian Andreetta¹, Vivien Begot², Jost Berthold³, Martin Elsman³, Troels Henriksen³, Maj-Britt Nordfang⁴ and Cosmin E. Oancea³
¹Nordea Capital Markets, ²Lexifi, HIPERFIT, Department of ³Computer Science and of ⁴Mathematical Sciences, University of Copenhagen

cbegot@nordea.com, vivien.begot@lexifi.com, [berthold,mael,athas]@diku.dk, mbnordfang@math.ku.dk, cosmin.oancea@diku.dk

Abstract—Commodity many-core hardware is now mainstream, driven in particular by the evolution of general purpose graphics programming units (GPGPUs), but parallel programming models are lagging behind in effectively exploiting the available application parallelism. There are two principal reasons. First, real-world applications often exhibit a rich composition of nested parallelism, whose statical extraction requires a set of (compiler) transformations that are tedious to do by hand and may be beyond the capability of the common user. Second, the best optimization strategy, with respect to what to parallelize and what to sequentialize, is often sensitive to the input dataset, and as such, it may require several code versions, maintained and optimized differently for different classes of datasets.

This paper studies three such array-based applications from the financial domain, which are suitable for GPGPU execution. For each application, we (i) describe all available parallelism via nested map-reduce functional combinators, (ii) describe the invariants and code transformations that govern the main trade-offs of a rich, dataset-sensitive optimizations space, and (iii) report target CPU and GPGPU code together with an evaluation that demonstrates optimization trade-offs and other difficulties. Finally, we believe this work provides useful insight into the language constructs and compiler infrastructure capable of expressing and optimizing such applications, and we report in-progress work in this direction.

I. INTRODUCTION

With the mainstream emergence of many-cores architecture, e.g., GPGPUs and Xeon Phi, massive parallelism has become a focus area of industrial application development. However, parallel-programming models are lagging behind the advance in hardware: parallelism extraction and optimization is still tedious and often requires specialized users.

Proposed solutions span a wide range of language and compiler techniques. On one end of the spectrum, we find the parallel assembly of our time, low-level APIs like CUDA, OPENCL, and OPENACC. On the opposite end are compilation techniques to automatically extract and optimize parallelism—usually within a language context, such as flattening [2] (NESL), polyhedral frameworks [3] (C), or inter-procedural summarization of array subscripts [4], [5] (Fortran). The use of domain-specific languages (DSLs) for parallel computation represents a middle-ground (with blurred boundaries), providing high-level operations with parallel implementations, and targeting data-parallel applications on arrays or graphs [6]–[10].

The HIPERFIT research center [1] set out to better exploit potential parallelism in one of the most computationally challenging domains: financial computations. The conjecture of the center is that domain-specific invariants should be exploited and encapsulated in suitable DSLs for best performance, requiring exploration of real-world applications. In this context, we have worked specifically on software to price financial derivatives, which we propose as a suite of benchmarks for parallelizing compilers and DSLs for parallelism.

The benchmark suite comprises three large-scale modeling components of a financial modeling engine, based on sequential source code provided by HIPERFIT partners. It includes (i) a pricing engine for financial contracts, (ii) a local-volatility calibration that reduces to solving many independent PDE systems via Crank-Nicolson’s finite differences method [11], and (iii) an interest-rate calibration based on a set of known swaption prices, whose results are input to the pricing engine.

The benchmark suite provides sequential (original) source code, ranging from hundreds to (couple of) thousands of lines of compact code, and different parallel versions for CPUs and GPGPUs. For example, the sequential code can be used to test auto-parallelization solutions, with the parallel versions providing the comparison baseline. In addition, we believe it is useful also to provide other material, including (i) simple code in a functional programming language, which fully specifies the available parallelism in terms of nested map-reduce operations on lists/arrays, (ii) documentation of important trade-offs that govern the optimization space, and (iii) customizable data sets to demonstrate these trade-offs. The rationale is that (parallel) application benchmarks are often heavily optimized towards specific hardware setups and generations, and are often run on friendly data sets, obscuring sub-optimal generality and potential opportunities to further optimization.

The provided benchmark programs can be described as a deeply-nested composition of data-parallel array operators (map, reduce, scan, and filter), within non-trivial control flows, such as dependent loops in which the optimal parallelization strategy is sensitive to the input data set. Such programs provide a compelling application for optimizing compilers and many-core hardware, including GPGPUs, albeit they present two major challenges to current solutions.

This work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HIPERFIT [1]). CA contributed to this article in his personal capacity. The views expressed are his own and do not necessarily represent the views of Nordea.

¹ On commodity hardware, the data-parallel GPGPU execution is several tens to several thousands of times faster than the sequential CPU execution.
First, supporting nested parallelism is important because, while companies are eager to reap the benefits of many-core architectures, they are typically not willing to rewrite their (sequential) code base more than once, and only if the resulting code structure still resembles the original algorithm. Furthermore, static extraction and optimization of parallelism (e.g., for GPGPU computations) requires the application of a set of compiler transformations, such as flattening, fusion, fission, loop interchange, and tiling, which are tedious, often beyond the capabilities of the common user, and typically result in an unmaintainable code base.

Second, at least to some extent, all solutions employ a “one size fits all” technique that results in one target program for all datasets. For example, (i) in an OpenCL program the user explicitly specifies what is to be executed in parallel and what sequentially, (ii) in a purely-functional context, the flattening transformation exploits all available parallelism and offers work-depth asymptotic guarantees but does not optimize memory-usage or locality of reference, while (iii) imperative solutions typically optimize the common case (maps and locality of reference) but without providing asymptotic guarantees.

The provided benchmark suite uses (realistic) customizable datasets to better explore the entire optimization space and reveal opportunities for combining the benefits of functional and imperative approaches. For instance, the dataset determines the parallelism degree of each level of a loop nest and effective hardware utilization may require either full parallelization or efficient sequentialization of that level (i.e., moderate flattening). This aspect requires solutions that generate specialized optimized code for the different cases and guard the different versions, ensuring sufficient parallelism and load-balancing.

As an important example, the scan operation, a well-known basic block of parallel programming [12], appears rarely (if at all) in the parallel version of SPEC programs. While difficult to recognize, and not always efficient to exploit, we found that instances of (segmented) scan are predominant in our benchmark. For example, the tridiagonal solver (TRIDAG) appears as a fully dependent loop, but it can be rewritten (via compiler techniques) into scans in which the associative operator is linear-function composition and $2 \times 2$ matrix multiplication, respectively. These scan instances are expensive to parallelize, yielding $\sim 6x$ instructional overhead and $\log n$ depth, but are necessary for two out of three data sets in order to fully utilize hardware and increase the speedup.

Finally, this work is an example of “the journey being more important than the destination”: Systematic parallelization of the original sequential programs, written in languages such as C and OCaml, has required us to (i) fully understand and express the available parallelism and (ii) execute a tedious, step-by-step transformation of the code bases in order to statically extract parallelism and to perform various (memory-related) optimizations. This process has provided important insights into identifying (i) the most useful language constructs that can express particular program patterns and (ii) a useful sequence of compiler transformations that can exploit the rich, dataset-sensitive optimizations space (i.e., efficient optimization of the common case, but within work-depth asymptotic guarantees.) Work is in progress for developing such a language (Futhark) and compiler infrastructure [13]–[16].

In conclusion, this work presents an application benchmark suite that constitutes a challenging and compelling application for optimizing compilers because a systematic hand-based implementation is simply too tedious. Main contributions are:

- A complete description of the available parallelism in three big-compute applications, suitable for GPGPUs.
- A detailed analysis of the invariants and (two-way) program transformations that implement the main tradeoffs of a rich, data-set-sensitive optimization space. Examples include moderate flattening, fusion vs. fission, and strength-reduction vs. independent computation.
- Insight into the language constructs and compiler infrastructure capable of effectively expressing and optimizing parallelism (deeply nested in non-trivial control flow).
- Parallel target CPU/GPU code, together with an empirical evaluation that demonstrates the tradeoffs and difficulties.

We hope that our financial real-world benchmarks complement common benchmark practice in the compiler community, and especially that the additional documentation and code prove useful for performance comparison across boundaries of language and programming model.

II. Preliminaries

This section presents the motivation for studying the three financial application, together with a brief characterization of each, and briefly introduces Futhark, the functional language used to describe the available application parallelism and code transformations; a detailed description of Futhark and its compiler infrastructure is found elsewhere [13]–[16].

A. Financial Motivation

The financial system is facing fundamental computational challenges lead by an increase in complexity, interconnectedness, and speed of interaction between participants. Financial institutions relocate capital across economic sectors, and are instrumental in providing stable growth. Should a large institution face liquidity shortage, a set of cascade effects may negatively impact the whole system. The impact of capital allocation across a large number of forecasted scenarios is estimated via large-scale simulations. For regulatory purposes, some of these scenarios involve critical conditions, which further increase computational demands.

These big-compute problems however, present a compelling and challenging application for commodity many-core hardware (e.g., GPGPUs), albeit they transcend the domain of embarrassingly parallel computing. For example, Monte Carlo simulations, originally developed to investigate the stochastic behavior of physical systems in complex, multidimensional spaces, have emerged as tool of choice in critical financial applications like risk modeling and contract pricing.

At top level, gains and risks can be described by means of a probabilistic formulation of possible market scenarios, estimated and aggregated with a Monte Carlo method, and evaluated at present time by a discount function.

This paper presents three components used in practice to implement such a financial mechanism:
Type Syntax

Types \( \exists \tau ::= \text{ bool } | \text{ char } | \text{ int } | \text{ real } \) / basic types
| \((\tau_1, ..., \tau_n)\) | [\(\tau\)] / tuples and regular arrays

SOAC Types

\[\oplus ::= (\alpha, \alpha) \to \alpha / \text{binary associative operator}\]

- replicate : \(\langle\text{int}, \alpha\rangle \to \alpha\)
- iota : \(\text{int} \to \text{int}\)
- zip : \((\alpha_1, ..., \alpha_n) \to (\alpha_1, ..., \alpha_n)\)
- unzip : \((\alpha_1, ..., \alpha_n) \to (\alpha_1, ..., \alpha_n)\)
- map : \((\alpha \to \beta), \alpha\) \to \beta\]
- zipWith : \(((\alpha_1, ..., \alpha_n), \beta), (\alpha_1, ..., \alpha_n) \to \beta\]
- filter : \((\alpha, \text{bool}), \alpha\) \to \alpha\]
- reduce : \(((\alpha, \alpha) \to \alpha), \alpha, \alpha) \to \alpha\]
- scan : \(((\alpha, \alpha) \to \alpha), \alpha, \alpha) \to \alpha\]

SOAC Semantics

- replicate \(\langle n, \alpha \rangle \equiv (\alpha, ..., \alpha) / \text{array of outer size n}\)
- iota \(\langle n \rangle \equiv (0, ..., n - 1)\)
- zip \(\langle \alpha_1, ..., \alpha_n \rangle \equiv (\alpha_1[0], ..., \alpha_n[0], \alpha_1[1], ..., \alpha_n[1], ...)\]
- unzip \(\equiv \text{zip}^{-1}\)
- map \((f, \alpha) \equiv (f \alpha[0], f \alpha[1], ...)\)
- zipWith \((f, \alpha_1, ..., \alpha_n) \equiv (f \alpha_1[0], ..., \alpha_n[0], f \alpha_1[1], ..., \alpha_n[1], ...))\]
- filter \((f, \alpha) \equiv \langle i | f \alpha[i] = \text{True}\rangle\]
- reduce \((\oplus, n, \alpha) \equiv (\alpha_1 \oplus \alpha_0 \oplus \alpha_1 | ... \oplus \alpha[n])\]
- scan \((\oplus, n, \alpha) \equiv (\alpha_1 \oplus \alpha_0 | (\alpha_1 \oplus \alpha_0 | \alpha_1)) | ... | \alpha[n])\]

Fig. 1. Types & Semantics of array constructors & second-order combinators.

- Section III presents a pricing engine for a set of vanilla and exotic options in scenarios with constant volatility.
- Section IV presents a local-volatility calibration, in which market volatility is modeled as a parameter of the option-call price. The volatility is calibrated by solving a system of continuous partial differential equations, using Crank-Nicolson’s finite differences method [17].
- Section V presents a calibration of the parameters of an interest-rate model, based on a set of available swap rates. The interest rate will be used to discount other financial products.

B. Futhark Language

Futhark is a mostly-monomorphic, statically typed, strictly evaluated, purely functional language that is primarily intended as a compiler intermediate representation (IL). It uses a syntax resembling SML [18]. It supports let bindings for local variables, but unlike SML, user-defined functions are monomorphic and their return and parameter types must be specified explicitly. Figure 1 presents the types and semantics of (some of) the built-in polymorphic, higher-order functions (SOACs) that can be used to construct and combine arrays:

- types include char,bool,int,real, tuples, and multi-dimensional regular arrays, i.e., all (sub-array) elements have identical shapes. An array of tuples is invalid if its translation to tuple of arrays results in an irregular array.
- iota and replicate are array constructors, and are typically used to create a normalized iteration space or to initialize an array. transpose is a special case of a more general operator that can interchange any two dimensions of an array (specified as int values).

The do-loop is essentially syntactic sugar for a certain form of tail-recursive function call: for example, denoting by \(t\) the type of \(x\), the loop in Figure 2 has the semantics of the function call on the right side. It is used by the user to express certain sequential computations that would be awkward to write functionally, and it enables several key lower-level optimisations, such as loop interchange and dependency analysis.

III. Option Pricing Benchmark

The presentation is organized as follows: Section III-A presents the main components of option pricing and shows that the benchmark translates directly to a nested map-reduce function composition, which expresses well both the algorithmic structure and the available parallelism. Section III-B investigates some of the high-level invariants and tradeoffs that govern the optimization space such as fusion and strength reduction.

Section III-C compares against an imperative setting. First, it identifies several key imperative-code patterns, such as scans, that would seriously hinder parallelism detection. Second, it validates the Futhark design, as it seems capable
to describe all available parallelism implicitly and to express dependent loops with in-place updates (which would be challenging in Haskell for example). Finally, Section III-D presents the empirical evaluation: (i) the sequential, multi-core, and GPGPU running times and (ii) the impact of various high- and low-level optimizations, such as fusion and memory coalescing.

A. Functional Basic Blocks of the Pricing Engine

Option contracts are one of the most common instruments exchanged between two financial actors. They are formulated in terms of a set of: (i) trigger conditions on market events, (ii) mathematical dependencies over a set of assets, named underlyings of the contract, and (iii) exercise dates, at which time the insurer holding a payoff, whose value depends on the temporal evolution of the underlyings. Two key components are necessary for the appreciation, at the current time, of the future value of the underlyings. Two key components are necessary for the future value of the underlyings. Two key components are necessary for the appreciation, at the current time, of the future value of the underlyings. Two key components are necessary for the appreciation, at the current time, of the future value of the underlyings.

- A stochastic description of the underlyings, which allows exploring the space of possible trigger events and payoff values, at the specified exercise dates. The parts of this component are described in more details in the remainder of this section.
- A technique to efficiently estimate the expected payoff by aggregating over the stochastic evolution of the underlyings. This component uses the quasi-random Monte Carlo method [21], which, in simple terms, averages over a population of prices obtained by regular, equi-distant sampling.

The function `mcPricing`, shown in Figure 3, represents a map-reduce implementation of the algorithm, the types of the main components and the manner in which they are composed. Its first four arguments correspond to the number of Monte Carlo iterations (n), the number of trigger dates (d), the number of underlyings (u), and the number of (different) market scenarios (m). The result of `mcPricing` is (semantically) a vector of size m (in \( \mathbb{R}^m \)) containing the presently-estimated prices for the current contract in each of the m market scenarios.

The implementation of `mcPricing` translates directly to a nest of mathematical function compositions. The outermost level is a `reduce` over a map composition applied to \( [1..n] \in \mathbb{Z}^n \). The map corresponds to the stochastic exploration and results in a matrix of prices (i.e., in \( \mathbb{R}^{n \times m} \)), in which the n rows and m columns correspond to the Monte Carlo iteration and the market scenarios, respectively. The reduce implements the Monte Carlo aggregation by adding componentwise \((\text{zipWith}(\text{op}+))\) the n price vectors produced by the map. The neutral element is a vector of m zeros \((\text{replicate}(m,0.0))\), and the result belongs to \( \mathbb{R}^m \).

The middle level corresponds to the implementation of the outermost map function parameter, and is (semantically) the composition of five functions: First, the stochastic exploration proceeds by drawing samples from an equi-probable, homogeneous distribution implemented via the Sobol multidimensional quasi-random generator [22]. This step corresponds to the `sobolIndR` call, which produces a pseudo-random sequence of size \( u \cdot d \) when applied to an integer `num`[1..n]. Second, the uniform samples are mapped by quantile-probability inversion [23] to Normally distributed values, later used to model the value of each underlying at the exercise dates. This mapping is performed by the function `ugaussian`, which has type \([0,1] \rightarrow \mathbb{R}^d\). Third, since the market is assumed to present good liquidity and no discontinuities, the probability distributions of the underlyings are independently modeled, with good approximation, as Brownian motions [24], continuous stochastic processes whose increments follow Normal distributions. The Brownian Bridge, denoted `brownBridge`, scales the resulted samples along the date dimension, independently for each underlying, in order to impose, also on non-observed dates, the properties of the stochastic processes [25]. It follows that the input vectors and the result are reshaped to \( u \times d \) matrices in which correlation is performed among dates (within each row).

Finally, the innermost three maps estimate the contract price for each of the m market scenarios (the result is in \( \mathbb{R}^m \)):

1. To express the expected correlation among underlyings, `blackScholes` scales once again the input samples via Cholesky composition by means of a positive-definite correlation matrix [26], which is part of `md_blsh`.
2. The obtained samples now mimic a (particular) market scenario and are provided as input to the `payoff` function that (i) computes the future gain from the contract in the current scenario and (ii) estimates the impact of the aggregated future payoff at present time via a suitable market discount model [25].
3. The obtained prices are divided by n (the Monte-Carlo space size) such that the average will result by summation.

We conclude with two remarks: First, while in Figure 3 array sizes are just provided as comments, Futhark infers (optimized) code that computes precise shapes at array creation points [16], whose runtime overhead is negligible in most cases. Second, Futhark borrows the expressiveness of Bird-Marteen’s formalism for specifying parallelism and high-level invariants, which are the subject of the next section.

---

\(^2\) The other (array) arguments of `mcPricing` are invariant to the stochastic exploration and are used in various stages of the algorithm. For example, `sob_bits` and `sob_dir_vcts` are the number of bits of a Sobol integer and Sobol’s direction vectors, `bb_data` are parameters of the Brownian Bridge, and `md_blsh` and `md_payof` are the parameters of m market scenarios, such as volatility, discount, and so on.
B. High-Level Invariants: Fusion and Strength Reduction

One important performance tradeoff refers to **fusion vs. fission**, and corresponds to two well-known invariants [19]: Map-map fusion (fission) states that mapping the elements of an array with a function and then the result with another function is equivalent to mapping the original array with the composition of the two functions: \((\text{map } g) \circ (\text{map } f) \equiv \text{map } (g \circ f)\). The second invariant states that a map-reduce composition can be rewritten to an equivalent form in which the input array is split into number-of-processors arrays of equal sizes, on which each processor performs the original computation sequentially, and finally, the local results are reduced in parallel:

\[
(\text{red } e) \circ (\text{map } f) \equiv (\text{red } e) \circ (\text{map } ((\text{red } e) \circ (\text{map } f))) \circ \text{dist}_p
\]

For option pricing, the direction in which these invariants should be applied to maximize performance is sensitive to the input dataset. For example, the outermost map and reduce in Figure 3, corresponding to the Monte Carlo exploration and aggregation, respectively, can be fused via equation 1:

If the memory footprint of map iterations, proportional with \(u \cdot d \cdot m\), fits in the GPGPU’s fast memory and the outermost degree of parallelism is sufficient to fully utilize the GPGPU then (i) slow (global) memory accesses are eliminated from the critical path, hence (ii) the execution behavior becomes compute rather than memory bound, and (iii) the memory consumption is reduced asymptotically (not proportional to \(n\)).

Otherwise, it is better to execute map and reduce as separate parallel operations and furthermore to distribute the outer map across the composed functions (map fission). On the one hand, this strategy allows for exploiting more parallelism, for instance, the inner maps of degree \(m\) in Figure 3 and the inner parallelism of each component (function). On the other hand, the distributed kernels are simpler, which relaxes the register pressure and increases the hardware utilization.

We note that Futhark supports both kinds of fusion at every level in the nest, even when the produced array is consumed in several places, without duplicating computation. This is achieved via a T2 graph-reduction technique [14], which is semantically applied to the program data-dependency graph, and which fuses map, filter, and reduce sequences into a parallel, more general construct named redomap.

The second important tradeoff refers to a **closed form vs. strength reduction** invariant that appears in the computation of Sobol sequences. We first explain the Sobol algorithm that translates directly to expressive Futhark code, and finally discuss the tradeoff.

A Sobol sequence [22] is an example of a quasi-random sequence of values \([x_0, x_1, \ldots, x_n, \ldots]\) from the unit hypercube \([0, 1]^d\). Intuitively, this means that any prefix of the sequence is guaranteed to contain a representative number of values from any hyperbox \(\prod_{j=1}^d [a_j, b_j]\), so the prefixes of the sequence can be used as successive better-representative uniform samples of the unit hypercube. Sobol sequences achieve a low discrepancy \(O(\log n / n)\). The Sobol algorithm for \(s = 1\) starts by choosing a primitive polynomial \(p\) over a Galois Field and by computing a number of direction vectors \(m_k\) by a recurrent formula that uses \(p\)'s coefficients. Each \(m_k\) is a positive integer and there are as many \(k\) as bits in the integer representation \(\text{num_bits}\).

The \(i^\text{th}\) Sobol number \(x_i\) can be computed independently of the others with the formula \(x_i = \bigoplus_{k \geq 1} B(i)_k \cdot m_k\), where \(B(i)_k\) denotes the value of the \(k^\text{th}\) bit of the canonical bit representation of the positive integer \(i\), and \(\oplus\) denotes the exclusive-or operator. In the above formula, one can use the reflected binary Gray code of \(i\) (instead of \(i\)), which is computed by taking the exclusive or of \(i\) with itself shifted one bit to the right. This modification to the algorithm changes the sequence of numbers produced but does not affect their asymptotic discrepancy. Using Gray codes enables a strength reduction opportunity, which results in a recurrent, more efficient formula \(x_{i+1} = x_i \oplus m_c\) for Sobol numbers, where \(c\) is the position of the least significant zero bit of \(B(i)\). The integer results are transformed to real numbers in \([0, 1]\) by division with \(2^{\text{num_bits}-1}\).

Finally, a Sobol sequence for \(s\)-dimensional values can be constructed by \(s\)-ary zipping of Sobol sequences for \(1\)-dimensional values, but it requires \(s\) sets of direction vectors (i.e., \(m_{i,k}\), where \(0 \leq i < s\) and \(0 \leq k < \text{num_bits} - 1\)).

Figure 4 shows Futhark code that expressively translates the independent and recurrent formulas, named sobolInd and sobolRec, respectively. The former filters the indices in \(0..\text{num_bits}-1\) that correspond to the set bits of the Gray code of \(i\), then maps each of the \(s = u \cdot d\) sets of direction vectors with a map-reduce function: the direction vector’s values corresponding to the filtered indices are retrieved and then reduced with the exclusive-or (xor) operator, denoted \(\oplus\). The index filtering can be seen as an optimization that reduces the number of xor operations on average by a factor of \(2\cdot s\), albeit at the cost of irregular nested parallelism and additional branches. This might be ill-advised on GPGPUs due to branch divergence overhead, and especially if we would like to exploit this inner level of parallelism. Fortunately this optimization can be easily reverted (by user or compiler) by fusing the filter producer on line 5 with the

---

3 The nomenclature is somewhat misleading since a quasi-random sequence is not truly pseudorandom: it makes no claim of being hard to predict.

4 We do not explain this step because this computation is not on the critical path, that is, direction vectors are computed once and used many times.
map-reduce consumer on lines 7–8. Such fusion would result in a GPU-efficient segmented reduce operation, because all segments have constant (warp) size num_bits=32.

The recurrent formula is implemented by sobolRec in Figure 4: (i) the call to recM selects for each direction vector the element at the index of the least significant zero of i, and (ii) the result is XORed component-wise (zipWith (op ^)) with the previous Sobol sequence (received as parameter).

The tradeoff refers to which formula to use for computing n consecutive Sobol numbers: The independent formula can simply be mapped, hence it enables efficient parallelization of depth $O(1)$, but requires up to 32× more work than the recurrent formula. The latter depends on the previous Sobol sequence, hence its parallelization requires first to map array [1..n] with function recM, then to scan the result with vectorized XOR, which exhibits less-efficient parallelism of $O(\log n)$ depth. Our solution combines the advantages by efficiently sequentializing the excess of application parallelism: the iteration space 1..n is strip-mined and chunks are processed in parallel, but the work inside a chunk is sequentialized. This is achieved by using the work-expensive independent formula for the first element of the chunk, thus enabling $O(1)$-depth parallelism, and amortizing this overhead by using the efficient strength-reduced formula for the remainder of the chunk. Here the data-sensitive input is n, the Monte-Carlo space size, which determines (i) the excess of parallelism (chunk size) and thus (ii) how well the overhead can be amortized.

C. Comparison with the Imperative Setting

Figure 5 shows the original, imperative pseudocode that computes n Sobol sequences under the (i) independent and (ii) recurrent formulas. Both versions exhibit significant hindrances for automatic or user identification of parallelism.

The loop using the independent formula (lines 2–12) can be parallelized by privatizing the arrays sobol and inds, which corresponds to proving that each potential read from an array is covered by a previous, same-iteration write into the same array. For sobol this is "easy" to prove by both user and compiler, because all its elements are first set to

```
// C(++) Sobol Independent Formula
1. int inds[num_bits], sobol[u*d], sob_dirs[u*d,num_bits];
2. for(i=0; i<num_bits; i++) { // outermost loop
3.  int len = 0, gcode = grayCode(i);
4.  for (j=0; j<num_bits; j++) {
5.    if (testBit(gcode, j)) {inds[len] = j; len++}
6.  }
7.  for (j=0; j<u*d; j++) {
8.    sobol[j] = 0;
9.  }
10. for (k=0; k<len; k++) {
11.    sobol[j] = sobol[j] ^ sob_dirs[j,inds[k]];}
12. // array inds (sobol): difficult (simple) to privatize
13. } // ... rest of fused code
```

The loop using the recurrent formula (lines 13–21) shows one of the many forms in which the scan primitive hides in imperative dependent loops: Here, memory use is optimized by recording only the current element of the scan, which is updated in a reduction pattern a a @ b, except that in the rest of the code, a is used outside reduction statements, which results in very sequential code. Another code pattern for scan is for (. *.) res[k] := res[k-1] @ b, which has a cross-iteration dependency of distance 1. This pattern appears in Figure 6 (line 9) and corresponds to a scan (zipWith (op *)) . In particular the first loop is a map, but the parallelism of the rest of the four-level nest is significantly more difficult to understand than its functional counterpart, which is fully described by nested map-reduce parallel operators and translates naturally to the Black Scholes formula.

There are however code patterns and code transformations

```
// C(++) Black Scholes // formula
1. real res[n,d,u], res[k], sd, lw, rw;
2. for(i=1; i<n; i++) { // outermost map
3.  res = res + (i-1)*d+u;
4.  for(int k=0; k<d; k++) {
5.    res = res + (i-1)*d+u;
6.    if (testBit(gcode, j)) {inds[len] = j; len++}
7.  }
8.  sobol[j] = sobol[j] ^ sob_dirs[j,inds[k]];}
9.  res[k,j] = res[k-1,j] + tmp;
10. // map (scan (zipWith ()))
```
that are more suitably expressed or reasoned about in imperative rather than functional notation. The first example refers to loops in which an array’s elements are produced (updated) and consumed across iterations. These loops are not infrequent, but are typically awkward to express functionally in a way that complies with the expected (in-place update) cost guarantees, and have motivated Futhark’s “imperative” constructs. Figure 7 shows such Futhark code that maps a normally-distributed sequence of size \(d \times u\) to Brownian bridge samples of dimension \(u \times d\). Since this step introduces dependencies between the column of the sample matrix, \(g\)auss is transposed and mapped with an unnamed (\(\lambda\)) function that uses a loop with in-place updates to model the cross-iteration dependencies on \(res\), which is declared as the only loop-variant variable (other than loop counter \(i\)). The loop uses three indirect arrays and each new element of \(res\) uses two other elements, of statically-known indices, produced in previous iterations. Still the parallel (\(map\)) and sequential (\(loop\)) code compose naturally.

The second case refers to low-level optimizations that rely on subscript analysis. In this context, Futhark’s imperative constructs makes it possible to represent, within the same language, lower-level representations of a program. For example, the outer \(map\) in Figure 7 can be turned into the parallel loop shown below, where all indices are now explicit and the \(result\) array, initialized with zeros, is computed inside the loop in transposed form \(d \times u\):

```futhark
let resT = copy( replicate(d, replicate(u, 0.0)) ) in
let gauss = reshape( (d,u), gauss ) in //res, gauss ∈ \mathbb{R}^{d \times u}
loop(res) = for p < u doall
  let resT'[bi[i+1]-1, p] = ... resT'[li[i+1]-1, p] = ...
    ... gauss[ i+1, p ] ...
  in res in res
```

Assuming GPGPU execution of the outer loop, and intra-thread (sequential) execution of the inner loop, the transposition of \(res\) is an optimization that ensures coalesced access to GPGPU global memory: Previously, each \(GP\)PU thread was computing a whole row of \(res\), and as such, a number of consecutive threads were accessing in one SIMD instruction elements of \(res\) with a stride \(d\). Transposition has moved the thread index \(p\) in the innermost subscript position, such that now consecutive threads access consecutive global-memory locations in each SIMD. Our experiments show that coalescing global accesses via transposition (or loop interchange) is one of the most impactful optimizations.

### D. Empirical Evaluation

The evaluation uses three datasets: The \textit{small} dataset uses \(n = 8388608\) Monte Carlo iterations to evaluate a vanilla-European call option: a contract with one exercise date, in which the payoff is the difference, if positive, between the value of a single underlying (Dj Euro Stoxx 50) at exercise date and a constant strike, which was set at issue date.

Options with multiple exercise dates may also force the holder to exercise the contract before maturity, in case the underlyings crossed specific barrier levels before one of the exercise dates. The \textit{medium} dataset uses \(1048576\) iterations to evaluate a discrete barrier contract over 3 underlyings, namely the area indexes Dj Euro Stoxx 50, Nikkei 225, and S&P 500, where a fixed payoff is a function of 5 trigger dates.

Finally, the \textit{large} dataset uses \(1048576\) iterations to evaluate a barrier option that is monitored daily, that is, \(367\) trigger dates, and in which the payoff is conditioned on the barrier event and the market values of the underlying at exercise time. The underlyings are the area indexes Dj Euro Stoxx 50, Nikkei 225, and S&P 500. The option pricing is run on two systems:

\textbf{H1} is an Intel(R) system, using 16 Xeon(R) cores, model E5-2650 v2, each supporting 2-way hardware multithreading and running at 2.60 GHz. \textbf{H1} is also equipped with a GeForce GTX 780 Ti NVIDIA GPGPU which uses 3 Gbytes of global memory, 2880 CUDA cores running at 1.08 GHz, and 1.5 Mbytes of L2 cache.

\textbf{H2} is an AMD Opteron system, using 32 cores, model 6274, and running at 2.2 GHz. \textbf{H2} is also equipped with a GeForce GTX 680 NVIDIA GPGPU, which uses 2 Gbytes of global memory, 1536 CUDA cores running at 1.02 GHz, and 512 Kbytes of L2 cache.

While for presentation purposes our evaluation reports parallel speedups, rather than runtime, Table I shows the sequential-CPU runtime for each of the two systems \textbf{H1} and \textbf{H2}, and each of the three datasets, so that parallel runtimes can be determined. Figure 8 shows the speedup results obtained on \textbf{H1} (left) and \textbf{H2} (right) for each of the three datasets: CPU 32 refers to the speedup of the parallel multi-core execution. GPU FUSE refers to the GPGPU execution of the fused version of the code, which executes in parallel only the Monte-Carlo iteration and aggregation, and does not accesses global memory on the critical path. As long as the local arrays are \textit{small}, this strategy yields significant speedup in comparison to GPU VECT, which corresponds to the distributed version of the code. As the size of the local arrays increases, each core consumes more of the sparse fast memory to the result that GPU utilization decreases. The \textit{medium} dataset seem to capture the sweet point: from there on, GPU VECT is winning, because its kernels are smaller, hence use less registers, and can be better optimized, e.g., inner parallelism. Furthermore, the fused version cannot execute the large dataset, because there is not enough fast memory for each each thread to hold \(365 \times 3\) real numbers.

Finally, GPU WO SR and GPU WO MC measure the impact of strength-reduction and memory coalescing optimizations, respectively, by presenting the speedup obtained \textit{without} their application. Strength reduction tends to be important when the number of dates and underlyings is \textit{small} because in such cases the weight of the Sobol component in the total program work is high. Also, as the degree of parallelism decreases, so does the size of the chunk that amortizes an independent formula against the execution of chunk-size recurrent formulas; it follows that the \textit{large} dataset is better of without strength reduction. \textit{Finally}, memory coalescing achieves a speedup factor in the \(10 \times -20\times\) range and is the most impactful optimization that we have observed.
The presentation is organized as follows: Section IV-A briefly states the financial problem to be addressed and sketches its mathematical solution. Section IV-B presents the code structure and the sequence of imperative code transformations that are necessary to disambiguate and to extract the algorithmic parallelism under a form that can be efficiently exploited by the GPGPU hardware. At this stage we identify several recurrences that can be parallelized but are (i) beyond the knowledge of the common user and (ii) introduce (constant but) significant work overhead in comparison to the sequential code. Finally, Section IV-C shows parallel CPU and GPGPU runtimes and demonstrates the tradeoff between efficient sequentialization and aggressive parallelization that provides asymptotic guarantees.

### A. Financial & Mathematical Description

The pricing engine presented in (previous) Section III uses a Black-Scholes model, and as such is most suitable for in-the-money “vanilla” options, for which volatility can be assumed constant. Options where the relation between market price and behavior of underlyings is more complex, for example because they may have several payoff triggers, are more appropriately modeled by imposing their (local) volatility as a function of time and current level of underlyings: \( \sigma(t, S(t)) \). In the following, we will focus on the case where the underlying is an equity stock. When the stock pays no dividends, the stock can be modeled as a stochastic differential equation of form: \(^6\)

\[
dS(t) = r(t)S(t)dt + \sigma(t, S(t))S(t)dW(t) \tag{2}
\]

where \( r(t) \) is the risk-free rate, \( W(t) \) is a Wiener process representing the inflow of randomness, and the instantaneous volatility \( \sigma(t, S(t)) \) measures the randomness amplitude. This reduces to solving numerically the partial differential equation (PDE)

\[
\frac{\partial f}{\partial t}(x, t) + r(t)x + \frac{\sigma(x, t)}{2} \frac{\partial^2 f}{\partial x^2}(x, t) - r(t)f(x, t) = 0 \tag{3}
\]

for many instances of a generic \( \sigma \) function and selecting the one that best matches the evolution of prices. This section uses the material and notation from [17] to briefly recount the main steps involved in solving such an equation by the Crank-Nicolson’s finite differences method [11]. The problem is to find \( f : S \times [0, T] \to \mathbb{R} \), which solves the (generic) PDE:

\[
\frac{\partial f}{\partial t}(x, t) + \mu(x, t) \frac{\partial f}{\partial x}(x, t) + \frac{\sigma(x, t)}{2} \frac{\partial^2 f}{\partial x^2}(x, t) - r(x, t)f(x, t) = 0 \tag{4}
\]

with some terminal condition expressed in terms of a known function \( F \). In essence, volatility calibration reduces to solve the above equation for many instances of \( \mu, \sigma, r \). For simplicity we discuss the case when \( S = \mathbb{R} \), but the benchmark uses a two-dimensional discretization of the space, hence \( S = \mathbb{R}^2 \).

The system of partial differential equations is solved by approximating the solution with a sequence of difference equations, which are solved by sequentially iterating over the time discretization, where the starting point is the known terminal condition 5. Figure 9 shows two methods that use the same difference formula to approximate the space derivatives, but differ in how the time-partial derivative is chosen, and this results in very different algorithmic (work-depth) properties:

**The explicit method**, shown in Figure 9(a), uses a backward-looking difference approximation of the time derivative \( D_t f_{j,n} = (f_{j,n} - f_{j,n-1})/\Delta t \), where \( n \in 1..N \) and \( j \in 1..J \) correspond to the discretized time and space. This results in equation \( f_{j,n+1} = \alpha_j f_{j,n} + \beta_j f_{j,n+1} + \gamma_j f_{j,n+1} \) that computes directly the unknown values at time \( n+1 \) from the values at time \( n \). The latter are known since we move backward in time: from terminal condition \( T \) towards 0. While the space discretization can be efficiently, map-like
\[\Delta x = (x_j - x_i)/x, \quad \Delta t = (t_N - t_i)/N\]

\[D x f_{j,n} = f_{j+1,n} - 2f_{j,n} + f_{j-1,n}\]

\[D t f_{j,n} = f_{j,n+1} - f_{j,n}\]

\[f_{j,n-1} = a_j f_{j,n} + b_j f_{j-1,n} + c_j f_{j+1,n}\]

where, \(a_j, b_j, c_j\) are known, \(\forall j \in \{1...J\}\), and we aim to find \(f_{j,n-1}, \forall j\) and we aim to find \(f_{j,n}, \forall j\).

Space discretization trivially parallel:

- Requires solving TRIDAG every time iter:
  - non-trivial scan parallelism.
- Requires fine-grained time discretization.

\[N \gg J \Rightarrow \text{deep depth } O(N)!\]

\[N \sim J \Rightarrow \text{reduced depth } O(J \log J)!\]

Fig. 9. (a) Explicit and (b) Implicit Finite Difference Methods.

Fig. 10. (a) Original Code & (b) After Privatization/Array Expansion.

parallelized, the time series is inherently sequential by nature and results into deep algorithmic depth because numerical stability requires a fine discretization of the time \((N \gg J)\).

The implicit method, shown in Figure 9(b), uses a forward difference approximation for the time derivative, which results in an equation \(f_{j,n-1} = a_j f_{j,n-1} + b_j f_{j,n} + c_j f_{j+1,n-1}\), in which a linear combination of unknown values at time \(n\) result in the known value at time \(n+1\), hence it reduces to solving a tridiagonal system of equations (TRIDAG). The advantage of the implicit method is that it does not require particularly small time steps, but the parallelization of the tridiagonal solver is beyond the knowledge of the common user, albeit it is possible via scans with linear-function composition and two-by-two matrix multiplication (associative) operators (see next section). While the scan parallelism has depth \(O(\log J)\) for one time iteration, the time and space discretization have comparable sizes, and as such, the total depth may improve asymptotically to \(O(J \log J)\) in comparison to \(O(N)\), \(N \gg J\), of the explicit method. Finally, Crank-Nicolson combines the two approaches, does not require particularly small time steps, converges faster, and is more accurate than the implicit method, albeit it still results in a tridiagonal system of equations.

B. Code Structure and Transformations

Figure 10(a) shows in C-like pseudocode the original structure of the code that implements volatility calibration: The outermost loop of index \(k=0...|U|-1\) solves equation 4 in a number of market scenarios characterized by different parameters \(\mu, \sigma, \nu\). Here, the space is considered two-dimensional, \(S = \mathbb{R}^2\), and we denote the space discretization with \(i=0...M-1\) and \(j=0...N-1\), on the \(y\) and \(x\) axes, respectively.

The body of the loop is the implementation of the Crank Nicolson finite-difference method, and it is formed by two loop nests: The first nest initializes array \(\text{tmpRes}\) in all the points of the space discretization. The second nest corresponds to the time series that starts from the terminal condition 5 and moves towards time 0 (i.e., \(t=T-1...0\)). At each time point \(t\), both the explicit and implicit method are combined to compute a new result based on the values obtained at previous time \(t+1\). In the code, this is represented by reading all the data in \(\text{tmpRes}\) corresponding to time \(t+1\) and later on updating \(\text{tmpRes}\) to the new result of current time \(t\). This read-write pattern creates a cross-iteration flow dependency carried by the loop of index \(t\), which shows the inherently-sequential semantics of the time series. As a final step, after time 0 was reached, some of the points of interest of the space discretization are saved in array \(\text{res}\) (for each of the \(U\) different market scenarios).

The remainder of this section describes the transformations that prepare the code for efficient GPGPU execution. The first difficulty corresponds to the outermost loop of Figure 10(a), which is annotated as sequential, albeit the scenario exploration stage is semantically parallel. The reason is that the space for array \(\text{tmpRes}\) is expressed as two-dimensional, is reused across the iterations of the outer loop, and as such, it generates frequent dependencies of all kinds. (Note how easily imperative code may obfuscate parallelism.) In such cases parallelism can be recovered by privatization, a code transformation that semantically moves the declaration of a variable inside the target loop, thus eliminating all dependencies, whenever it can be proven that any read from the variable is covered by a previous write in the same iteration. In our case all elements of \(\text{tmpRes}\) are first written in the first nest, and then read and written in the time series (second nest). It follows that it is safe to move the declaration of \(\text{tmpRes}\) inside the outermost loop, and to mark the latter as parallel. However, working with local arrays is inconvenient (if not impossible) when aiming at GPGPU code generation, and this is when array expansion comes to the rescue. The declaration of \(\text{tmpRes}\) is moved outside the loop (to global memory) but it receives an extra dimension of size equal to \(U\), the outermost loop count. The array expansion preserves parallelism because each outer iteration \(k\) uses now its own \(M \times N\) subarray, recorded in \(\text{tmpRes}[k]\); the resulted code is shown in Figure 10(b).

The second difficulty relates to the GPGPU programming model being thought to exploit static, rather than dynamic parallelism. In our context, static parallelism would correspond to the structure of a perfect nest in which consecutive outer loops are parallel. For example, the code in Figure 10(b) exhibits significant nested parallelism, but for example the outermost and any of the inner loops cannot both be executed in parallel; while parallelism exist, it cannot be exploited! Perfect nests can be manufactured with two (important) transformations, namely loop interchange and loop distribution.
the legality of loop distribution in this case can be proven by relatively simple direction-vector dependence analysis.

For example, the recurrence $y_i = a_i + b_i x_i - c_i y_{i-1}$ can be brought to a parallel form by (i) first performing the change of variable $y_i \leftarrow q_{i+1}/q_i$, then (ii) normalizing the obtained equation resulting in $q_{i+1} = a_i q_{i+1} + b_i q_i$, then (iii) adding a trivial equation to form a system of two equations with two unknowns, which can be computed for all $[q_{i+1}, q_i]$ vectors as a $2 \times 2$ matrix-multiplication (associative) operator:

$$
\begin{bmatrix}
q_{i+1} \\
q_i
\end{bmatrix} =
\begin{bmatrix}
a_i & b_i \\
1.0 & 0.0
\end{bmatrix}
\begin{bmatrix}
q_i \\
q_{i-1}
\end{bmatrix} =
\begin{bmatrix}
a_i & b_i \\
1.0 & 0.0
\end{bmatrix}
\begin{bmatrix}
a_i & b_i \\
1.0 & 0.0
\end{bmatrix} \cdots
\begin{bmatrix}
a_1 & b_1 \\
1.0 & 0.0
\end{bmatrix}
\begin{bmatrix}
x_{0} \\
1.0
\end{bmatrix}
$$

Similarly, recurrence $x_i = a_i + b_i * x_{i-1}$ can be computed by a $\text{scan}$ with a linear-function composition operator (which is clearly associative). However, exploiting TRIDAG’s parallelism comes at a cost: it requires six $\text{map}$ operations and three $\text{scan}$s, which, in comparison to the sequential algorithm, exhibits significant (constant-factor) work overhead, both in execution time and in terms of memory pressure. This overhead strongly hints that it is preferable to sequentialize $\text{tridag}$ efficiently if there exists enough parallelism at outer levels.

C. Empirical Evaluation

The evaluation uses three contrived datasets: (i) $\text{small}$ has $U \times M \times N \times T = 16 \times 32 \times 256 \times 256$ and is intended to be friendly with the aggressive approach that parallelizes TRIDAG, (ii) $\text{medium}$ has $U \times M \times N \times T = 128 \times 32 \times 256 \times 256$ and is intended to be a midpoint, and (iii) the $\text{large}$ dataset has

Fig. 11. After (a) Outer-Loop Distribution & (b) Interchange & Distribution.
TRIDAG enabling efficient sequentialization of parallelism in the two outer loops to utilize the hardware, while the code, i.e., it does not need to exit to the CPU, in the sense that the current implementation restricts that takes advantage of all parallelism, including the one of parallelizes the outermost loops, but efficiently sequentializes referred to parallel execution on 32

Table II. Sequential CPU Runtimes on Systems H1 and H2.

<table>
<thead>
<tr>
<th>Dataset/Machine</th>
<th>Large</th>
<th>Medium</th>
<th>Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq CPU Runtime on H1</td>
<td>73.8 sec</td>
<td>8.5 sec</td>
<td>4.3 sec</td>
</tr>
<tr>
<td>Seq CPU Runtime on H2</td>
<td>141.4 sec</td>
<td>20.2 sec</td>
<td>10.1 sec</td>
</tr>
</tbody>
</table>

Fig. 13. Speedup with respect to Sequential Execution on Systems H1 & H2
CPU32: execution on 32 (multicore) hardware threads.
GPU Out denotes GPGPU execution exploiting only outer (map) parallelism.
GPU All denotes GPGPU execution exploiting all parallelism (also TRIDAG)

$UXMNXT=128 \times 256 \times 256 \times 256$ and contains enough parallelism in the two outer loops to utilize the hardware, while enabling efficient sequentialization of TRIDAG.

Similar to section III-D, which also describes the H1 and H2 hardware used for testing, we report parallel speedups, rather than runtime, but specify in Table II the sequential-CPU runtime for each of the two systems, and each of the three datasets, so that parallel runtimes can always be determined. Figure 13 shows the speedup results obtained on the two systems by three different code versions: (i) CPU32 refers to parallel execution on 32 hardware threads, (ii) GPU Out refers to a version of code that executes on GPU and parallelizes the outermost loops, but efficiently sequentializes TRIDAG and (iii) GPU All refers to the version of code that takes advantage of all parallelism, including the one of TRIDAG. Note however that GPU ALL is relatively efficient in itself, in the sense that the current implementation restricts the values of M and N to multiples of 2 less or equal to 256. The consequence is that a scanned segment never crosses the kernel’s block boundaries, which means that scan can execute entirely in local memory and can be fused with the rest of the code, i.e., it does not need to exit to the CPU.

As expected, (i) GPU Out is significantly faster than GPU ALL when enough outer parallelism is available, because it efficiently sequentializes TRIDAG, and (ii) GPU ALL is performing much better on small datasets, where the other method utilizes the hardware parallelism poorly.

V. Interest Rate Model Benchmark

The presentation is organized as follows. Section V-A gives motivation for the importance of interest rate modeling and motivation for calculating results quickly on large input data sets using big computations, based on Monte Carlo Simulation techniques. Section V-B presents the main parts of a two-factor mean-reversion interest-rate model that includes both a pricing component and a calibration component, which in essential ways make use of the associated pricing component. For this benchmark, we shall not describe the code fragments in details as for the first two benchmarks, but, in Section V-C, we present an empirical evaluation of the benchmark by providing sequential, multi-core, and GPGPU running times.

A. Financial Motivation

The interest rate is the premium paid by a borrower to a lender. It incorporates and measures the market consensus on risk-free cost of capital, inflationary expectations, cost of transactions, and the level of risk in the investment. The interest rate is a function of time, market and financial instrument, and can be used to value at present time the future value of the assets under scrutiny. Financial derivatives based on interest rates (such as swaps) are among the largest groups of derivatives exchanged on the global markets [25].

The important role of interest-rate models in financial computations has become even more central with recent regulatory dispositions like Basel III [29], requiring financial institutions to report financial portfolios in different market scenarios. Some of these scenarios may include credit events linked to the solidity of major counterparties. These requirements, by not necessarily assuming both counterparties as solvable for the entire lifetime of a contract, may induce discontinuities in the contract obligations. The consequence of these events has to be estimated at the future time of discontinuity, and correctly priced at present time for it to be reported to auditing authorities.

Before being employed, an interest-rate model has to be calibrated. Its independent parameters have to be estimated on the current market conditions, so that future scenarios evolve from an observed market state. It is therefore paramount for financial institutions to choose an interest-rate model that is fast to compute, and a calibration process that is robust to market noise.

B. Financial Description

The next paragraphs describe the interest-rate model and the calibration process, which requires (i) an interest-rate model, (ii) a reference dataset, with data observed in the market, (iii) a bounded parameter space, (iv) an error function measuring the divergence between the reference dataset and the output of a model based on a specific set of parameters, and (v) a search strategy for the identification of the optimal parameter set.

The interest rate model object of this benchmark is the short-term two-additive-factor Gaussian model (G2++), developed by Brigo and Mercurio [30], to whom we refer for a more detailed exposition. This model, while of speed comparable to a single-factor model like Hull-White [25], is more robust and has been shown to react quickly to market volatilities.

The G2++ model describes the interest rate curve as a composition of two correlated stochastic processes. The model...
is a function of five independent parameters, which, once calibrated according to the present market consensus, can be employed in valuations. Each process is a random walk (Brownian motion) described by a mean reversion constant $(a, b)$ and two volatility terms $(\sigma$ and $\eta)$. The two Brownian motions are correlated by a constant factor $\rho$. At time $t$, the interest rate $r_t$ can be expressed as $r_t = x_t + y_t + \phi_t$, with the stochastic processes

$$dx_t = -ax_t dt + \sigma dW^1_t \quad dy_t = -by_t dt + \eta dW^2_t$$

correlated by $pdt = dW^1_t dW^2_t$. The third term, $\phi_t$, is deterministic in time $t$ and is defined by the following equation:

$$\phi_t = f^M(0,T) + \frac{\sigma^2}{2a^2} \left(1 - e^{-at}\right)^2 + \frac{\eta^2}{2a^2} \left(1 - e^{-bt}\right)^2 + \rho \frac{\sigma \eta}{a^2} \left(1 - e^{-at}\right) \left(1 - e^{-bt}\right)$$

with offset $f^M(0,T)$ depending on the instantaneous forward rate at time 0 for the maturity $M$.

The five independent parameters $\text{param} = (\alpha, \beta, \sigma, \eta, \rho)$ of the G2++ model influence the individual behavior of its two stochastic processes, and their correlation. With the $\text{param}$ tuple describing the current market, an interest rate profile can be constructed for the most likely future scenarios. Additionally, an interest rate model calibrated on a portion of a market can be used to price other instruments in the same market.

A European interest-rate swaption is a contract granting its owner the right, but not the obligation, to enter into an underlying swap with the issuer [25]. This right is dependent on the level of the interest rate at the expiration date of the swaption. A swap is a contract in which two counterparties exchange a proportion of the cash flows of one party’s financial instrument (e.g., a fixed-rate loan) for those of the other party’s financial instrument (e.g., a floating-rate loan, or a fixed-rate loan in a different currency). The contract allows the one party to access advantageous loan conditions in its own market and hedge monetary fluctuations by sharing its risk with another party’s comparative advantage in a different capital market.

Our reference dataset, capturing the market consensus on the interest rate, consists of 196 European swaption quotes, with constant swap frequency of 6 months and maturity dates and swap terms ranging from 1 to 30 years. The calibration process identifies a set of $\text{param}$ tuples most likely to describe the current market (concretely, the swaption quotes). Since an inverse analytical relation between $\text{param}$ and market contracts is not available, the calibration is a search over a continuous 5-dimensional parameter space. The parameter space is rugged, so that minor updates in the $\text{param}$ tuple would produce quite different interest-rate scenarios. For the search to be efficient, an effective exploration of the parameter space is necessary, as well as a quick relation between some market contracts and the $\text{param}$ tuple. Brigo and Mercurio [30] have indicated a numerical relation between the price of European swaption contracts and the G2++ parameters. From an algorithmic perspective, a set $p$ of proposals of candidate $\text{param}$ values is generated. Subsequently, with a Markov Chain Monte Carlo variation of the Differential Evolution search, a portion of the population $p$ is discarded, and replaced with new computed elements.

The Markov Chain Monte Carlo Differential Evolution (DE-MCMC) is an heuristic search over a continuous space [31]. It is a kind of genetic algorithm, where a population of solution vectors $x$ is measured by a fitness function $\pi(\cdot)$. At each step of the algorithm, a portion of the population is discarded, and the discarded elements are replaced with a recombination (cross-over) depending on an algebraic combination of two surviving candidates. The speed of the search and the coverage of the space can be tuned with the choice of the ratio between the surviving and discarded subpopulations. DE-MCMC is a population MCMC algorithm, in which multiple chains are run in parallel, and DE suggests appropriate scale and orientation for the transitions in the chain. In a statistical context, it is important to estimate both an uncertainty over an optimal solution returned by DE, and the amount of clusters of optimal solutions. Both goals can be obtained by a Bayesian analysis using a Markov Chain Monte Carlo (MCMC) simulation.

For each candidate $\text{param}$, all swaptions are priced according to the G2++ model, as described by Brigo and Mercurio [30, Chapter 4]. An error function $\text{Err}(\text{param})$ summarizes the differences between the observed marked prices and the $\text{param}$-modeled prices for the proposed model $p$, which can be accepted or rejected according to the quality of $\text{Err}(\text{param})$. We shall not describe the swaption pricing and the $\text{Err}$ function in more detail here, but mention that the heart of the pricer involves the use of Brent’s root-finding method [32, Chapter 4] and the computation of an integral using the Gauss–Hermite quadrature technique [33].

### C. Empirical Evaluation

The evaluation uses currently only one dataset. Sequential CPU timings are presented in Table III. The results for the
parallel speedup on the two systems H1 and H2 are shown in Figure 14. This application is quite challenging and tedious to parallelize: First it exhibits irregular parallelism, with frequent segmented reduce operators on irregular arrays, nested inside convergence loops. If the irregular parallelism is not exploited, for example via a moderate flattening-like transformation, it leads to significant load imbalance, thread divergence, etc. Our technique has been to "pad" the irregular arrays such that segments do not cross local blocks, which allows the segmented reduce to be fused with the rest of the kernel. While in this case the GPU execution is couple of times faster than the parallel CPU execution, the speedup is significantly lower than the one achieved by volatility calibration and nowhere near the ones understood.

A large body of related work includes the work on embedded domain specific languages (DSLs) for programming massively parallel architectures, such as GPGPUs. Initial examples of such libraries include Nikola [58], a Haskell library for targeting CUDA. Later work includes the Accelerate [59] and Obsidian [60] Haskell libraries that, with different sets of fusion and optimization techniques, targets OpenCL and CUDA. An example of a more specialized language is SPL [10], which specifies and efficiently computes stochastic processes.

Probably most related to the work on Futhark is the work on SAC [61], which seeks to provide a common ground between functional and imperative domains for targeting parallel architectures, including both multi-processor architectures [62] and massively data-parallel architectures [63]. SAC uses with and for loops to express map-reduce style parallelism and sequential computation, respectively. More complex array constructs can be compiled into with and for loops, as demonstrated, for instance, by the compilation of the APL programming language [64] into SAC [65]. Compared to SAC, Futhark holds on to the SOAC combinators also in the intermediate representations in order to perform critical optimizations, such as fusion, even in cases involving filtering and scans, which are not straightforward constructs for SAC to cope with.

Also related to the present work is the work on array languages in general (including APL [64] and its derivatives) and the work on capturing the essential mathematical algebraic aspects of array programming [66] and list programming [19] for functional parallelization. Compilers for array languages also depend on inferring shape information either dynamically or statically [67], although they can often assume that the arrays operated on are regular, which is not the case for Futhark programs. Another piece of related work is the work on the FISH [68] programming language, which uses partial evaluation and program specialization for resolving shape information at compile time.

A scalable technique for targeting parallel architectures in the presence of nested parallelism is to apply Blelloch’s flattening transformation [69]. Blelloch’s technique has also been applied in the context of compiling NESL [70], but is sometimes incurring a drastic memory overhead. In an attempt at coping with this issue and for processing large data streams, while still making use of all available parallelism, a streaming version of NESL, called SNESSL has been developed [71], which supports a stream datatype for which data can be processed in chunks and for which the cost-model is explicit.

A final strand of related work is the work on benchmark suites, in particular for testing and verifying performance of hardware components and software tools. An important benchmark suite for testing accelerated hardware, such as GPGPUs and their related software tool chains is the SPEC ACCEL benchmark [72] provided by the Standard Performance Evaluation Committee (SPEC). Compared to the present work, the SPEC ACCEL benchmark contains few, if any, applications related to the financial domain and, further, the goal of the
SPEC ACCEL benchmark is not to demonstrate that different utilization of parallelism can be appropriate for different input data sets.

VII. CONCLUSION AND FUTURE WORK

We have presented three real-life financial benchmarks, based on sequential source code provided by HIPERFIT partners [1]. The benchmarks include (i) a generic option pricing benchmark, (ii) a volatility calibration benchmark, and (iii) an interest rate pricing and calibration benchmark. For the first two benchmarks, concrete code is presented and we have demonstrated opportunities for parallelization by presenting how the benchmarks can be expressed and transformed in Futhark, a parallel array language that integrates functional second-order array combinators with support for imperative-like array-update constructs. Empirical measurements demonstrate the feasibility of the proposed transformations and show that important optimizations applicable for some input data may not be applicable to other input data sets that are perhaps larger in size and do not fit appropriately in, for instance, GPGPU memory. Finally, we believe that the technique presented in this paper can be applied outside the finance domain, and future work will aim at integrating them in the Futhark compiler tool chain, which would ultimately result in efficient GPGPU code generation.

REFERENCES


[33] N. M. Steen, G. D. Byrne, and E. M. Gelbard, “Gaussian quadrature for the integrals \( \int_a^b \exp(-x^2) f(x) dx \) and \( \int_a^b \exp(-x^2)^2 f(x) dx \),” Math. Comp., vol. 23, pp. 661–671, 1969.


