A Financial Benchmark for GPGPU Compilation

Cosmin Oancea, Jost Berthold, and Martin Elsman
Department of Computer Science
University of Copenhagen
[cosmin.oancea,berthold,mael]@diku.dk

Christian Andreetta
Department of Computer Science
University of Bergen
christian.andreetta@cbu.uib.no

***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 advent of multicore architectures, and driven by the inventions of specialised accelerators such as Xeon Phi and computational GPGPUs, massive parallelism has become a focus area of industrial application development. New hardware developments, however, require tool developers to rethink and adapt the respective programming models: Commodity programming models are lagging behind and are unable effectively to utilize the available parallelism.

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]–[9].

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 system, based on sequential source code provided by HiperFIT partners. It includes (i) a pricing engine for financial contracts operating in liquid markets, (ii) a stochastic-volatility calibration where market volatility is modeled as a system of continuous partial differential equations, solved via Crank-Nicolson's finite differences method [10], 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[1]. 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, obfuscating 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.

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

This work has been partially supported by the Danish Council for Strategic Research under contract number 10-092299 (HiperFIT [1]).

[1] On commodity hardware, the data-parallel GPGPU execution is several tens to several thousands of times faster than the sequential CPU execution.

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 (`map`s 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 [11], 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 `scan`s in which the associative operator is linear-function composition and 2×2 matrix multiplication, respectively. These `scan` instances are expensive to parallelize, yielding $\sim 6\times$ instructional overhead and $\log n$ depth, but 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 [12]–[15].

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 [12]–[15].

A. Financial Motivation

The financial system is facing fundamental challenges because of their complexity, interconnectedness, and speed of interaction. Financial institution relocate capital across economical sectors, and are thus instrumental in providing a stable economic climate, but should they default, they may start a snowball effect with catastrophic results for the whole system. To protect against such effects, banking and insurance regulations require institutions to evaluate their reliability in a large number of economic scenarios, some of which present critical conditions and require in turn large-scale modeling.

These simulations are typically Big Compute problems that 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 by physicists to investigate efficiently the stochastic behavior of complex, multidimensional spaces, have emerged as the tool of choice in critical financial applications like risk modeling and contract pricing.

At top level, gains and risks are quantified by means of a probabilistic description of these (yet unknown) scenarios, and a (Monte Carlo) method to evaluate presently their economic impact. Risk management is performed by allocating capital according to the foreseen value of available investment opportunities and in the same time insuring against outcomes that would invalidate the strategy. This paper presents three components used in practice to implement such a mechanism:

- Section III presents a pricing engine for contracts operating in liquid markets,

Type Syntax

Types $\ni \tau ::=$ `bool` | `char` | `int` | `real` // basic types
| $(\tau_1, \ldots, \tau_n)$ | $[\tau]$ // tuples and regular arrays

SOAC ***Types***

$\oplus :: (\alpha, \alpha) \to \alpha$ // binary associative operator
replicate :: $(\texttt{int}, \alpha) \to [\alpha]$
iota :: $\texttt{int} \to [\texttt{int}]$
zip :: $([\alpha_1], \ldots, [\alpha_n]) \to [(\alpha_1, \ldots, \alpha_n)]$
unzip :: $[(\alpha_1, \ldots, \alpha_n)] \to ([\alpha_1], \ldots, [\alpha_n])$
map :: $((\alpha \to \beta), [\alpha]) \to [\beta]$
zipWith :: $(((\alpha_1, \ldots, \alpha_n) \to \beta), [\alpha_1], \ldots, [\alpha_n]) \to [\beta]$
filter :: $((\alpha, \texttt{bool}), [\alpha]) \to [\alpha]$
reduce :: $(((\alpha, \alpha) \to \alpha), \alpha, [\alpha]) \to \alpha$
scan :: $(((\alpha, \alpha) \to \alpha), \alpha, [\alpha]) \to [\alpha]$

SOAC ***Semantics***

replicate $(n, a) \equiv \{a, \ldots, a\}$ // array of outer size n
iota $(n) \equiv \{0, \ldots, n-1\}$
zip $(a_1, \ldots, a_n) \equiv \{(a_1[0], \ldots, a_n[0]), (a_1[1], \ldots, a_n[1]), \ldots\}$
unzip $\equiv$ **zip**$^{-1}$
map $(f, a) \equiv \{f(a[0]), f(a[1]), \ldots\}$
zipWith $(f, a_1, \ldots, a_n) \equiv \{f(a_1[0], \ldots, a_n[0]), f(a_1[1], \ldots, a_n[1]), \ldots\}$
filter $(f, a) \equiv \{a[i] | f(a[i]) = \texttt{True}\}$
reduce $(\oplus, n_e, a) \equiv (\ldots((n_e \oplus a[0]) \oplus a[1]) \ldots \oplus a[n])$
scan $(\oplus, n_e, a) \equiv \{n_e \oplus a[0], ((n_e \oplus a[0]) \oplus a[1]), \ldots\}$

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

- Section IV presents a stochastic-volatility calibration, in which market volatility is modeled as as a system of continuous partial differential equations, which are solved via Crank-Nicolson's finite differences method [16].
- Section V presents a calibration of the interest rate based on a set of available swaption prices.

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 [17]. 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. Futhark supports:

- basic types `char`, `bool`, `int`, `real`, tuples, and multi-dimensional *regular arrays*, that is, arrays for which all immediate subarrays have equal shapes. An array of tuple type is valid if and only if its translation to tuple of arrays results in regular arrays.
- `iota` and `replicate` array constructors, which are typically used to create a normalized iteration space or to initialize an array, and `transpose`, as a special case of a more general operator that can interchange any two dimensions of an array (specified as `int` values).
- the typical array operators of the Bird-Meertens Formalism (BMF) [18], which have inherently parallel semantics and include `zip`, `unzip`, `map`, `reduce`, `filter`, and `scan`. Anonymous (and curried) functions are syntactically permitted only as SOAC function arguments, as, for instance, in `let t =... in map(fn int(int x)=>x+t, iota(t))`.

```
loop (x = a) =                    fun t f(int i, int n, t x) =
  for i < n do          ⇒           if i >= n then x else f(i+1, n, g(x))
    g(x)
in body                           let x = f(0, n, a)
                                  in body
```

Fig. 2. A `do` loop has the semantics of a tail-recursive function call.

An important Futhark feature is that `zip` and `zipWith` accept an arbitrary number of (n-ary) array arguments that must have the same outer-dimension size. The `zip` and `zipWith` constructs represent syntactic sugar: Arrays-of-tuples are compiled to a tuple-of-array representation with the SOAC operators (i.e., `map`, `reduce`, `filter`, and `scan`) becoming n-ary operators.

Finally, Futhark supports two imperative-like constructs: `do` loops and `let-with` bindings, which are useful for expressing sequential computation that updates *in place* the array's elements, e.g., loops exhibiting cross-iteration dependencies on arrays. A `let-with` binding uses the syntax

`let b = a with [`$\texttt{i}_1$`,..,`$\texttt{i}_k$`] <- v in body`

and has the semantics that `body` is evaluated with `b` bound to the value of `a`, except that the element at position (`i`$_1$`,..,`$\texttt{i}_k$) is updated to the value of `v`. Such an update passes the type-checker if, intuitively, it can be proven that `a` (and its aliases) are not used on any execution path following the update, and is supported without losing referential transparency via a mechanism resembling uniqueness types [12], [13], [19]. It follows that an in-place update takes time proportional to the total size of `v`, rather than of `a`. When the source and destination array share the same name, we provide the syntactic sugar notation: `let a[`$\texttt{i}_1$`,..,`$\texttt{i}_k$`] = v in body`.

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 `scan`s, 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

```
fun [real] mcPricing(                                    //ℝ^m
    int n, int d, int u, int m, int contract, int sob_bits,
    [[int ]] sob_dirs,    ([[int ]], [[real]])      bb_data,
    [( [[real]], [[real]], [[real]], [real] )]      md_blsch,
    [( [real], [real] )]                            md_payof) =
  let prices = map (          //ℝ^{n×m}
      fn [int] (int sn) =>    //ℕ → ℝ^m
        let num = sn + 1 in
      in let sob=sobolIndR(sob_bits,sob_dirs,num)//ℤ → [0,1)^{u·d}
      in let gau=ugaussian(sob)          //[0,1)^{u·d}  →  ℝ^{u·d}
      in let bbr=brownBridge(u,d,bb_data,gau)//ℝ^{u·d} → ℝ^{u×d}
      in let bsh=map(blackScholes(u, bbr),md_blsch)//ℝ^{m×u×d}
      in let pay=zipWith(payoff(contract),bsh,md_payof)//ℝ^m
             in map( op /(toReal(n)), pay) //ℝ^m
    , iota(n) ) //{0..n-1}∈ ℤ^n

  in reduce(zipWith(op+),replicate(m,0.0),prices)//ℝ^{n×m} → ℝ^m
```

Fig. 3. Pricing Engine: Functional Basic Blocks and Types.

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 insuring actor will reward the option holder with 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 contract:

- 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 exploration. This component uses the quasi-random population Monte Carlo method [20], which, in simple terms, averages the obtained prices.

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[2] 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 ∘ 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 (`zipWith(op+)`) the `n` price vectors produced by the `map`. The neutral element is a vector of `m` zeros (`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 [21]. This step corresponds to the `sobolIndR` call, which produces a pseudo-random sequence of size $u \cdot d$ when applied to an integer `num`$\in$`[1..n]`. Second, the resulted uniform samples are mapped by quantile-probability inversion [22] to Normally distributed values, which model the value of each underlying at the exercise dates. This mapping is performed by the function `ugaussian`, which has type $[0,1)^{u\cdot d} \rightarrow \mathbb{R}^{u\cdot d}$. 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 continuous stochastic processes that follow Normal distributions, that is, Brownian motions [23]. The Brownian Bridge, denoted `brownBridge`, scales the resulted samples along the date dimension, and independently for each underlying, in order to extend the conservation of the stochastic-processes properties also to non-observed dates (to preserve modeling consistency [24]). It follows that the input vectors and the result are reshaped to `u×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 [25], which is part of `md_blsch`.
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 [24].
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 [15], 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.

B. High-Level Invariants: Fusion and Strength Reduction

One important performance tradeoff refers to ***fusion vs. fission***, and corresponds to two well-known invariants [18]: Map-map fusion (fission) states that mapping the elements of an array

[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_blsch` and `md_payof` are the parameters of `m` market scenarious, such as volatility, discount, and so on.

with a function and then the result with another function is equivalent to mapping the original array with the composition of the two functions: $(\texttt{map g}) \circ (\texttt{map f}) \equiv \texttt{map}\,(\texttt{g} \circ \texttt{f})$. 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:

$$(\texttt{red} \oplus e) \circ (\texttt{map f}) \equiv (\texttt{red} \odot e) \circ (\texttt{map}\,((\texttt{red} \odot e) \circ (\texttt{map f}))) \circ \texttt{dist}_p \quad (1)$$

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·d·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 `map`s 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 [13], 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 [21] is an example of a quasi-random[3] sequence of values $[x_0, x_1, \ldots, x_n, \ldots]$ from the unit hypercube $[0,1)^s$. Intuitively, this means that any prefix of the sequence is guaranteed to contain a representative number of values from any hyperbox $\prod_{j=1}^{s}[a_j, b_j)$, 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 $O(\frac{\log^s n}{n})$. 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[4]. Each m_k is a positive integer and there are as many ks as bits in the integer representation (`num_bits`).

[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.

```
1. fun int grayCode(int x) = (x >> 1) ^ x // ^ denotes xor
2. fun bool testBit(int n, int ind) =
3.     let t = (1 << ind) in (n & t) == t // & is bitwise And

   // Sobol Independent Formula; sob_dirs ∈ N^(u·d)×num_bits
4. fun [int] sobolInd(int bits_num,[[int]] sob_dirs,int i)=
5.     let inds = filter(testBit(grayCode(i)), iota(num_bits))
6.     in map( fn int ([int] dir_vct) =>
7.               let row = map(fn int(int i)=>dir_vct[i], inds )
8.               in     reduce( op ^, 0, row )
9.          , sob_dirs )
   //the first n Sobol numbers can be computed with:
   //map( sobolInd(num_bits,sob_dirs), [0..n-1] )

   // Sobol Strength-Reduced (Recurrent) Formula
10.fun [int] sobolRec([[int]] sob_dirs, [int] prev, int i)=
11.    let col = recM(sob_dirs, i) in zipWith(op^,prev,col)

12.fun [int] recM( [[int]] sob_dirs, int i ) =
13.    let bit= index_of_least_significant_0(i) in
14.    map( fn int([int] row) => row[bit], sob_dirs )
   //the first n Sobol numbers can be computed with:
   //(scan(zipWith(op^), [0..0], map(recM(sobol_dirs),[1..n]))
```

Fig. 4. Futhark Code for Computing Pseudo-Random Sobol Sequences.

The i'th Sobol number x_i can be computed independently of the others with the formula $x_i = \bigoplus_{k \geq 0} B(i)_k \cdot m_k$, where $B(i)_k$ denotes the value of the k'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 $2^{\texttt{num\_bits}-1}$.

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 $0 \leq k < \texttt{num\_bits} - 1$).

Figure 4 shows Futhark code that expressively translates the independent and recurrent formulas, named `sobolInd` and `sobolRec`, respectively. The former `filter`s the indices in `0...num_bits-1` that correspond to the set bits of the Gray code of `i`, then `map`s each of the s = `u*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 `op ^`. 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 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

```
    // (i) C(++) Sobol Independent Formula
1.  int inds[num_bits], sobol[u*d], sob_dirs[u*d,num_bits];
2.  for(i=0; i<n; 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.      for (k=0; k<len; k++) {
10.       sobol[j] = sobol[j] ^ sob_dirs[j,inds[k]];
11.   } } // ... rest of fused code
12. } // array inds (sobol): difficult (simple) to privatize

13. //(ii) C(++) Sobol Strength-Reduced (Recurrent) Formula
14. int sobol[u*d], sob_dirs^T[num_bits,u*d];
15. for(i=1; i<=n; i++) { // outermost map
16.     int bit = index_of_least_significant_0(i);
17.     for (j=0; j<u*d; j++) {
18.         sobol[j] = sobol[j] ^ sob_dirs^T[bit,j];
19.     }
20.     ... code using array named sobol ...
21. } // parallelizable via map-scan: difficult to recognize
```

Fig. 5. C-like Pseudocode for Computing Sobol Sequences.

the element at the index of the least significant zero of `i`, and (ii) the result is `xor`ed 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 `map`ped, hence it enables efficient parallelization of depth $O(1)$, but requires up to $32\times$ 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 `0` at line `8` and then repeatedly read-written in the loop at lines `9-11`. At least for the compiler, however, privatization of `inds` is anything but easy, because dependency analysis on arrays typically requires subscripts to be affine formulas in the loop indices, but the array update `inds[len]=j` at

```
   // C(++) Black Scholes // formula?
1. real res[n,d,u], ...;
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.    for(int j=0; j<u; j++) {
6.      real tmp = 0.0;
7.      for (l=0; l<=j; l++){ tmp += md_cs[j,l]*deltas[k,l]; }
8.      tmp = exp( tmp*md_vols[k,j] + md_drifts[k,j] );
9.      res[k,j] = res[k-1,j] * tmp;
10.     // map (scan (zipWith (*)))
   } } }
```

Fig. 6. C-like Pseudocode for Cholesky-Composition Scaling (Black-Scholes).

```
fun [[real]] brownBridge ( int u, int d, ([[int ]],[[real]])
                                          bb_data, [real] gauss ) =
  bb_inds ∈ ℕ^{3×d}, bb_coef ∈ ℝ^{3×d}, gauss ∈ ℝ^{d·u}
  let (bb_inds,bb_coef) = bb_data in
  let (bi, li, ri) = (bb_inds[0], bb_inds[1], bb_inds[2]) in
  let (sd, lw, rw) = (bb_coef[0], bb_coef[1], bb_coef[2]) in

  map( fn [real] ([real] gau) =>  // λ ∈ ℝ^d → ℝ^d
         let res = copy( replicate(d, 0.0) )   in
         let res[ bi[0] - 1 ] = sd[0] * gau[0] in
         loop(res) = for i < d-1 do
           let (j,k,l) = (li[i+1]-1, ri[i+1]-1,bi[i+1]-1) in
           let tmp = res[k]*rw[i+1] + sd[i+1]*gau[i+1]   in

           let res[l] = if( (j + 1) == 0) then tmp
                        else tmp + lw[i+1] * res[j]
           in  res
         in res
     , transpose( reshape((d,u), gauss)) )//map result ∈ ℝ^{u×d}
```

Fig. 7. Futhark "Imperative"-like Code for the Brownian Bridge.

line `5` does not comply with this restriction, that is, `len` cannot possibly be expressed as an affine formula in `j` because it is conditionally incremented inside the loop at lines `4-6`. While techniques to parallelize such loops exist [26], [27], they require complex analysis, which is frustrating given that the outer and inner loops at lines `2-12` and `4-6` are the imperative implementations of a `map` and a `filter`, which both have inherently parallel semantics.

The loop[5] using the recurrent formula (line `15-21` in Figure 5) 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(..k..) 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 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

[5]Please remember that this loop corresponds to a `map-scan` composition, shown in Figure 4 and discussed in the (previous) section III-B.

Dataset/Machine	Small	Medium	Large
Seq CPU Runtime on H1	0.8 sec	1.28 sec	11.23 sec
Seq CPU Runtime on H2	1.54 sec	2.14 sec	16.06 sec

TABLE I. SEQUENTIAL CPU RUNTIMES ON SYSTEMS H1 AND H2.

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·u to Brownian bridge samples of dimension u×d. Since this step introduces dependencies between the column of the sample matrix, gauss is transposed and mapped with an unnamed (λ) 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-unknown 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×u:

```
let res^T  = copy( replicate(d, replicate(u, 0.0)) ) in
let gauss = reshape( (d,u), gauss ) in //res, gauss ∈ ℝ^{d×u}
loop(res) = for p < u doall
  let res^T[bi[0]-1, p] = sd[0] * gauss[0, p] in
  loop(res) = for i < d-1 do
    let res^T[bi[i+1]-1, p] = ... res^T[ 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 GPGPU 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 *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 *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 *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:

- **H1** is an Intel(R) system, using 16 Xeon(R) cores, model E5-2650 v2, each supporting 2-way hardware multi-threading and running at 2.60 GHz. **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.
- **H2** is an AMD Opteron system, using 32 cores, model 6274, and running at 2.2 GHz. **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 **H1** and **H2**, and each of the three datasets, so that parallel runtimes can be determined. Figure 8 shows the speedup results obtained on **H1** (left) and **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 *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 *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 365×3 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 ***without*** their application. Strength reduction tends to be important when the number of dates and underlyings is *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 *large* dataset is better of without strength reduction. *Finally*, memory coalescing achieves a speedup factor in the $10\times - 20\times$ range and is the most impactful optimization that we have observed.

IV. STOCHASTIC VOLATILITY CALIBRATION

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

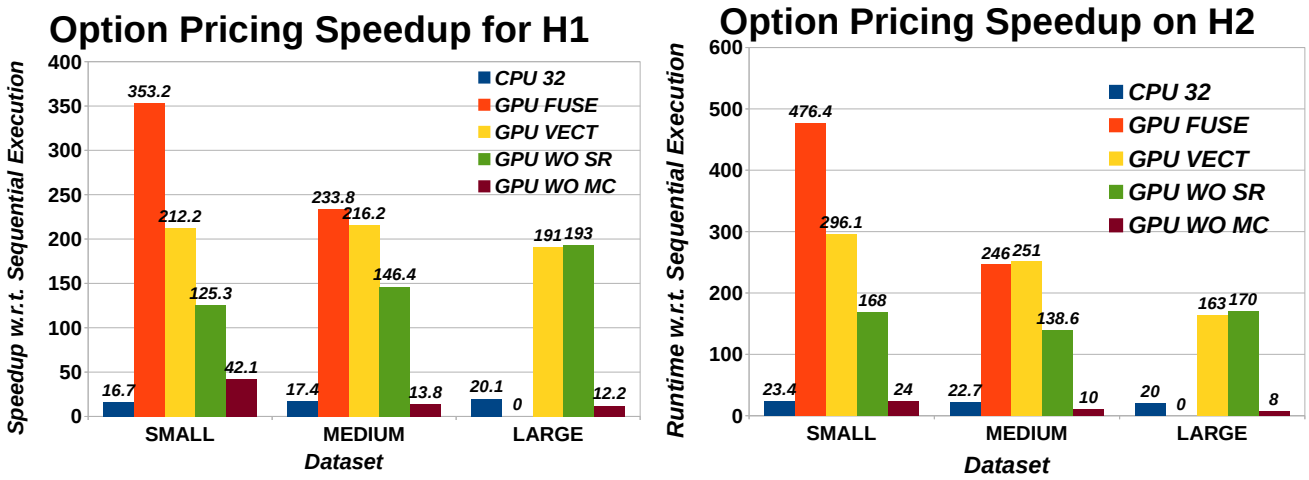

Fig. 8. Option Pricing Speedup with respect to Sequential Execution on Systems H1 (left) and H2 (right).
CPU32: execution on 32 (multicore) hardware threads.
GPU FUSE and GPU VECT denote the fused and distributed GPU code with all other optimizations ON, respectively.
GPU WO SR and GPU WO MC denote the speedup in the absence of strength reduction and memory coalescing, respectively.

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 benchmark presented in this section is an engine for stochastic volatility calibration. Given a set of observed prices of contracts, it identifies a model of such prices, as a function of (i) the unknown volatility, (ii) the known time and strikes, and (iii) some other unobserved parameters.

In this case, the volatility is modeled as a system of continuous partial differential equations, which are solved via Crank-Nicolson's finite differences method [10]. This section briefly recounts the main steps of the solution, using the material and notations from [16], where matters are presented in detail. The problem is to find a function $f(x,t)$, $f : S \times [0,T] \rightarrow \mathbb{R}$, which solves the second-order partial differential equation:

$$\frac{\partial f}{\partial t}(x,t)+\mu(x,t)\frac{\partial f}{\partial x}(x,t)+\frac{1}{2}\sigma(x,t)^2\frac{\partial^2 f}{\partial^2 x}(x,t)-r(x,t)f(x,t)=0 \quad (2)$$

$$f(x,T) = F(x), \text{ where } (x,t) \in S \times [0,T) \text{ and } F : S \rightarrow \mathbb{R} \quad (3)$$

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 μ, σ, 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 3. 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:

$$\Delta x = (x_J - x_1)/J,\ \Delta t = (t_N - t_1)/N$$
$$D_x f_{j,n} = \frac{f_{j+1,n}-f_{j-1,n}}{2\Delta x},\ D_x^2 f_{j,n} = \frac{f_{j+1,n}-2f_{j,n}+f_{j-1,n}}{(\Delta x)^2}$$

$$D_t^- f_{j,n} = \frac{f_{j,n} - f_{j,n-1}}{\Delta t} \qquad D_t^+ f_{j,n} = \frac{f_{j,n+1} - f_{j,n}}{\Delta t}$$

$$f_{j,n-1} = \alpha_{j,n} f_{j-1,n} + \beta_{j,n} f_{j,n} + \gamma_{j,n} f_{j+1,n} \qquad f_{j,n+1} = a_{j,n} f_{j-1,n} + b_{j,n} f_{j,n} + c_{j,n} f_{j+1,n}$$

where $f_{j,n}$ known $\forall j \in \{1 \ldots J\}$ and we aim to find $f_{j,n-1}$, $\forall j$ — where $f_{j,n+1}$ known $\forall j \in \{1 \ldots J\}$ and we aim to find $f_{j,n}$, $\forall j$

Space discretization trivially parallel: depth $O(N)$, work $O(J \cdot N)$, Requires fine-grained time discretization, $N \gg J \Rightarrow$ deep depth $O(N)$! (a)

Requires solving TRIDAG every time iter: non-trivial `scan` parallelism, Allows coarse-grain time discretization, $N \sim J \Rightarrow$ reduced depth $O(J \log J)$! (b)

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

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 $f_{j,n-1} = \alpha_{j,n} f_{j-1,n} + \beta_{j,n} f_{j,n} + \gamma_{j,n} f_{j+1,n}$ 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 parallelized, the time series is inherently sequential by nature and results into *deep* algorithmic *depth* because the mathematics requires particularly small time steps ($N \gg J$).

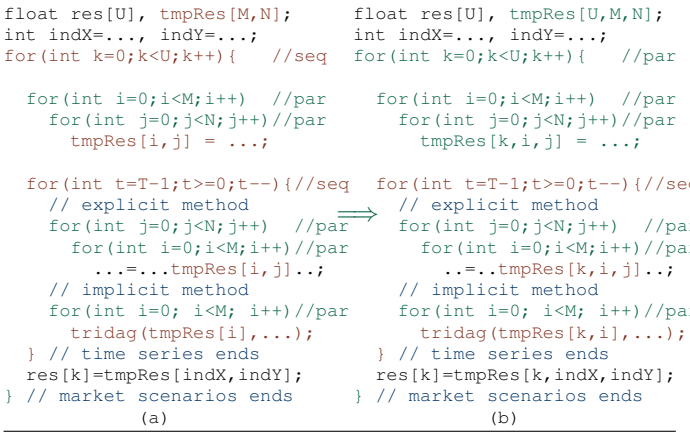

Fig. 10. (a) Original Code Structure & (b) After Privatization/Array Expansion.

Fig. 11. After (a) Outer-Loop Distribution & (b) Interchange & Distribution.

The implicit method, shown in Figure 9(b), uses a forward difference approximation for the time derivative, which results in equation $f_{j,n+1} = a_{j,n}f_{j-1,n} + b_{j,n}f_{j,n} + c_{j,n}f_{j+1,n}$, 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 `scan`s 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 2 in a number of market scenarios characterized by different parameters μ, σ, r. 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 is formed by two loop nests: The first nest initializes array `tmpRes` in all the points of the space discretization. The second nest corresponds to the time series that starts from the terminal condition 3 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 `tmpRes` corresponding to time `t+1` and later on updating `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 `res` (for each of the `U` different market scenarios).

The remainder of this section describes the sequence of transformations that prepares 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 `tmpRes[M,N]`, declared 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 `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 `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 `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×N` subarray, recorded in `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. While in general, determining the legality of the transformations

```
// INPUT:  a,b,c,r ∈ ℝ^n
// OUTPUT: x, y    ∈ ℝ^n

x[0] = r[0]; y[0] = b[0];
// Forward Recurrences
// identification requires
// forward substitution
// and loop distribution
for(i=1; i<n; i++) {
  float beta  = a[i]/y[i-1];
  y[i] = b[i] - beta*c[i-1];
  x[i] = r[i] - beta*x[i-1];
}
// Backward Recurrence
x[n-1] = x[n-1] / y[n-1];
for(i=n-2; i>=0; i--)
  x[i] = x[i]/y[i] -
         c[i]*x[i+1])/y[i];
```

(a)

```
// INPUT:  a,b,c,r ∈ ℝ^n
// OUTPUT: x, y    ∈ ℝ^n

x[0] = r[0]; y[0] = b[0];

// forward recurrences
for(i=1; i<n; i++)
  y[i]=b[i]-a[i]*c[i-1]/y[i-1];
⟹
// forward recurrence
for(i=1; i<n; i++)
  x[i]=r[i]-a[i]/y[i-1]*x[i-1];

// backward recurrence
x[n-1] = x[n-1] / y[n-1];
for(i=n-2; i>=0; i--)
  x[i] = x[i]/y[i] -
         c[i]*x[i+1])/y[i];
```

(b)

Fig. 12. After (a) Outer-Loop Distribution & (b) Interchange & Distribution.

is non-trivial (e.g., using direction-vector based dependence analysis), matters are simple for parallel loops; a parallel loop (i) can be safely interchanged inward in a nest, and (ii) can be safely distributed across its statements. Figure 11(a) shows the code resulting after the distribution of the outer loop. The initialization part is now an order-three perfect-loop nest that exhibits U×M×N degree of parallelism, and leads to efficient GPGPU utilization. However, for the second nest, the degree of exploitable parallelism has not yet been improved because the sequential time-series loop separates the outer and several inner parallel loops. Interchanging the outer and time-series loops, and then distributing again the (former) outer loop of index k results in the code shown in Figure 11(b), for which every iteration of the time series executes two GPGPU kernels with degree of parallelism U×M×N and U×M, respectively.

The third difficulty corresponds to memory coalescing. For example, the global access tmpRes[k,i,j] in the second nest in Figure 11(b) is uncoalesced; consecutive GPGPU threads correspond to the fastest-changing loop of index i, but they access, in the SIMD instruction tmpRes[k,i,j], array elements with a stride N (rather than 1). The simplest solution is to interchange the loops of indices i and j. While this interchange is effective for the shown code, it would not resolve a case where two accesses x[k,i,j] and y[k,j,i] happen in the same instruction, because the loop interchange will coalesce one access but uncoalesce the other. In such cases, array transposition has proven to be an effective solution, because the penalty of uncoalesced accesses is typically much higher than the (non-negligible) overhead of transposition.

Finally, one can observe in Figure 11(b) that the third kernel, which implements the tridiagonal solver (tridag), offers only two parallel dimensions (of depth $O(1)$): the loops of indices k and i. This combine to a U×M degree of parallelism that might be too small to efficiently utilize the hardware on some datasets. In such cases, the parallelization of the tridiagonal solver, which is discussed in the remainder of this section, might significantly improve matters.

Figure 12(a) shows the pseudocode that typically implements a tridiagonal solver. Both loops are sequential because the values of x[i] and y[i] in iteration i depends on the result of iteration i−1, that is, x[i−1] and y[i−1]. However, this awfully sequential loop can be automatically transformed to parallel code in four steps: (i) the local variable beta is forward substituted in both recurrences of the first loop, then (ii) the first loop is distributed across its two remaining statements[6] and (iii) the resulting one-statement recurrences, shown in Figure 12(a), are recognized as belonging to one of the "known" patterns: $x_i = a_i + b_i \cdot x_{i-1}$ or $y_i = a_i + b_i / y_{i-1}$, and finally, (iv) the loops are replaced with parallel code.

Dataset/Machine	Large	Medium	Small
Seq CPU Runtime on H1	73.8 sec	8.5 sec	4.3 sec
Seq CPU Runtime on H2	141.4 sec	20.2 sec	10.1 sec

TABLE II. SEQUENTIAL CPU RUNTIMES ON SYSTEMS H1 AND H2.

For example, the recurrence $y_i = a_i + b_i / y_{i-1}$ 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 $q_{i+1} = a_i \cdot q_{i-1} + b_i \cdot q_i$, 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 scan with a 2×2 matrix-multiplication (associative) operator:

$$\begin{bmatrix} q_{i+1} \\ q_i \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} =$$

$$\begin{bmatrix} a_i & b_i \\ 1.0 & 0.0 \end{bmatrix} * \begin{bmatrix} a_{i-1} & b_{i-1} \\ 1.0 & 0.0 \end{bmatrix} * \cdots * \begin{bmatrix} a_1 & b_1 \\ 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ 1.0 \end{bmatrix}$$

Similarly, recurrence $x_i = a_i + b_i * x_{i-1}$ can be computed by a scan with a linear-function composition operator (which is clearly associative). However, exploiting TRIDAG's parallelism comes at a cost: it requires six map operations and three scans, 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 tridag efficiently if there exists enough parallelism at outer levels.

C. Empirical Evaluation

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

[6] The legality of loop distribution in this case can be proven by relatively simple direction-vector dependence analysis.

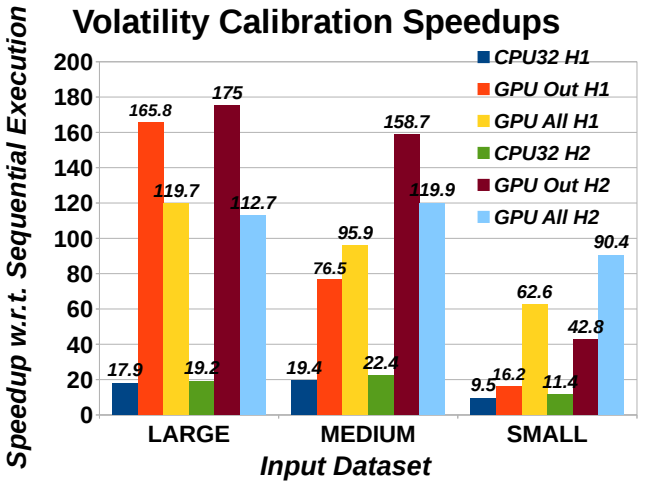

Fig. 13. Speedup with respect to Sequential Execution on Systems `H1` & `H2` `CPU32`: execution on 32 (multicore) hardware threads. `GPU Out` denotes GPGPU execution exploiting only outer (`map`) parallelism. `GPU All` denotes GPGPU execution exploiting all parallelism (also TRIDAG)

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, which again means that 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 valuate 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 [24].

The important role of interest-rate models in financial computations has become even more central with recent regulatory dispositions like Basel III [28], 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 [29], to whom we refer for a more detailed exposition. This model, while of speed comparable to a single-factor model like Hull-White [24], 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 (σ and η). The two Brownian motions are correlated by a constant factor ρ. At time t, the interest rate r_t can be expressed as $r_t = x_t + y_t + \phi_t$, with the stochastic processes

$$dx_t = -ax_t dt + \sigma dW_t^1 \qquad dy_t = -by_t dt + \eta dW_t^2$$

correlated by $\rho dt = dW_t^1 dW_t^2$. The third term, ϕ_t, is deterministic in time t and is defined by the following equation:

$$\phi_t = \begin{array}{l} f^M(0,T) + \frac{\sigma^2}{2a^2}\left(1-e^{-aT}\right)^2 + \\ \frac{\eta^2}{2b^2}\left(1-e^{-bT}\right)^2 + \rho\frac{\sigma\eta}{ab}\left(1-e^{-aT}\right)\left(1-e^{-bT}\right) \end{array}$$

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

The five independent parameters $\texttt{param} = (\alpha, \beta, \sigma, \eta, \rho)$ of the G2++ model influence the individual behavior of its two stochastic processes, and their correlation. With the `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 at 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 [24]. 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 loan 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 `param` tuples most likely to describe the current market (concretely, the swaption quotes). Since an inverse analytical relation between `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 `param` tuple would produce quite different interest-rate scenarios. For the search to be efficient, an efficient exploration of the parameter space is necessary, as well as a quick relation between some market contracts and the `param` tuple. Brigo and Mercurio [29] 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 `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 [30]. It is a kind of genetic algorithm, where a population of solution vectors $\mathbf{x}$ 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 ration 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 `param`, all swaptions are priced according to the G2++ model, as described by Brigo and Mercurio [29, Chapter 4]. An error function $Err(\texttt{param})$ summarizes the differences between the observed marked prices and the `param`-modeled prices for the proposed model p, which can be accepted or rejected according to the quality of $Err(\texttt{param})$. We shall not describe the swaption pricing and the *Err* function in more detail here, but mention that the heart of the pricer involves the use of Brent's root-finding method [31, Chapter 4] and the computation of an integral using the Gauss–Hermite quadrature technique [32].

Dataset/Machine	Dataset
Seq CPU Runtime on H1	510 sec
Seq CPU Runtime on H2	660 sec

TABLE III. SEQUENTIAL CPU RUNTIMES ON SYSTEMS H1 AND H2.

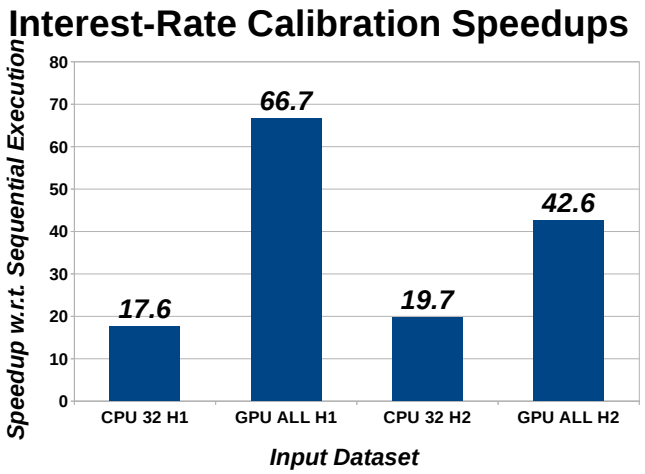

Fig. 14. Speedup with respect to Sequential Execution on System `H1` & `H2` `CPU 32`: 32 CPU hardware threads. `GPU ALL`: GPGPU's speedup.

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 achieved for generic pricing. These results seem to indicate that real-world applications have different characteristics and that some are more suited to parallel execution then others.

VI. RELATED WORK

There are several strands of related work. First, a considerable amount of work has been published on parallelizing

financial computations on parallel hardware, such as GPGPUs, reporting impressive speedups (see Joshi [33] or Lee et al. [34], for example) or focusing on production integration in large banks' IT infrastructure [35]. Our work differs in that we seek to express parallel patterns in the provided benchmarks as high-level functional constructs, with the aim of systematically (and automatically) generate efficient (and even data-dependent) parallel code. Futhark is an ongoing effort at providing a language tool chain that follows this strategic direction [12]–[15].

Second, a large body of related work is concerned with auto-parallelization of imperative code. Such work includes idiom-recognition [26] and static dependency analyses [36], [37] for examining and interchanging loop nests in order to recognize and achieve parallelism [3], but also to improve memory access patterns. Alternatively, more dynamic techniques can be applied, for example a variety of algorithms and transformations [38]–[40] aimed at enhancing the locality of reference by restructuring the data or the order in which the data is processed. Other techniques may track dependencies at runtime and may extract partial parallelism [41]–[43], but such techniques have not been evaluated (yet) on GPGPUs.

A third strand of related work covers techniques for achieving GPGPU high-performance [44], which involves (i) achieving memory coalescing via block-tiling, (ii) optimizing register usage via loop unrolling, and (iii) performing data prefetching for hiding memory latency. These techniques form the basis for application-specific hand-optimized high-performance GPGPU code, written in language frameworks such as CUDA or OpenCL, and for establishing high-performance GPGPU libraries, such as Thrust [45]. Implementation of these principles as compiler optimizations ranges from (i) heuristics based on pattern matching [46]–[48], over (ii) more formal modeling of affine transformations via the polyhedral model [49], [50], to (iii) aggresive techniques, such as loop collapsing, that may be applicable even for irregular control-flow and irregular memory access patterns [51].

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 [52], a Haskell library for targeting CUDA. Later work includes Accelerate [53] and Obsidian [54], Haskell libraries that, with different sets of fusion- and optimization techniques, targets OpenCL and CUDA.

Probably most related to the work on Futhark is the work on SAC [55], which seeks to provide a common ground between functional and imperative domains for targeting parallel architectures, including both multi-processor architectures [56] and massively data-parallel architectures [57]. 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 [58] into SAC [59]. 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 [58] and its derivatives) and the work on capturing the essential mathematical algebraic aspects of array programming [60] and list programming [18] for functional parallelization. Compilers for array languages also depend on inferring shape information either dynamically or statically [61], 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 [62] 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 [63]. Blelloch's technique has also been applied in the context of compiling NESL [64], 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 SNESL has been developed [65], 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 [66] 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 parallism 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.

There are several opportunities for future work. First, more work is needed in order for the Futhark compiler tool chain to generate efficient code for GPGPUs, based on the techniques described in this paper. Second, we believe that the techniques described here may be applied also outside of the finance domain.

REFERENCES

[1] J. Berthold, A. Filinski, F. Henglein, K. Larsen, M. Steffensen, and B. Vinter, "Functional High Performance Financial IT – The HIPERFIT Research Center in Copenhagen," in *Trends in Functional Prog. (TFP) – Revised Selected Papers*, ser. LNCS 7193. Springer, 2012, pp. 98–113.

[2] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a Portable Nested Data-Parallel Language," *Journal of parallel and distributed computing*, vol. 21, no. 1, pp. 4–14, 1994.

[3] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization," in *Procs. Sym. Principles of Prog. Lang. (POPL)*. ACM, 2011, pp. 549–562.

[4] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural Parallelization Analysis in SUIF," *Trans. on Prog. Lang. and Sys. (TOPLAS)*, vol. 27(4), pp. 662–731, 2005.

[5] C. E. Oancea and L. Rauchwerger, "Logical Inference Techniques for Loop Parallelization," in *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, 2012, pp. 509–520.

[6] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007.

[7] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic Galois: On-demand, portable and parameterless," in *Procs. Int. Conf. Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2014.

[8] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly, "Performance Analysis and Optimisation of the OP2 Framework on Many-core Architectures," *The Computer Journal*, 2011.

[9] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell Array Codes with Multicore GPUs," in *Procs. Decl. Aspects of Multicore Prog. (DAMP)*. ACM, 2011, pp. 3–14.

[10] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," *Math. Proc. of the Cambridge Philosophical Society*, vol. 43, pp. 50–67, January 1947.

[11] G. E. Blelloch, "Scans as Primitive Parallel Operations," *Computers, IEEE Transactions*, vol. 38, no. 11, pp. 1526–1538, 1989.

[12] T. Henriksen, "Exploiting functional invariants to optimise parallelism: a dataflow approach," Master's thesis, DIKU, Denmark, 2014.

[13] T. Henriksen and C. E. Oancea, "A T2 Graph-Reduction Approach to Fusion," in *Procs. Funct. High-Perf. Comp. (FHPC)*. ACM, 2013, pp. 47–58.

[14] T. Henriksen and C. E. Oancea, "Bounds Checking: An Instance of Hybrid Analysis," in *Procs. Int. Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[15] T. Henriksen, M. Elsman, and C. E. Oancea, "Size Slicing - A Hybrid Approach to Size Inference in Futhark," in *Procs. Funct. High-Perf. Comp. (FHPC)*. ACM, SIGPLAN, 2014.

[16] C. Munk, "Introduction to the Numerical Solution of Partial Differential Equations in Finance," 2007.

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[18] R. S. Bird, "An Introduction to the Theory of Lists," in *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*, 1987, pp. 5–42.

[19] E. Barendsen and S. Smetsers, "Conventional and Uniqueness Typing in Graph Rewrite Systems," in *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*, ser. LNCS, vol. 761, 1993, pp. 41–51.

[20] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. New York: Springer, 2004.

[21] P. Bratley and B. L. Fox, "Algorithm 659 Implementing Sobol's Quasirandom Sequence Generator," *ACM Trans. on Math. Software (TOMS)*, vol. 14(1), pp. 88–100, 1988.

[22] M. Wichura, "Algorithm AS 241: The percentage points of the Normal distribution," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 37, no. 3, pp. 477–484, 1988.

[23] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *The Journal of Political Economy*, pp. 637–654, 1973.

[24] J. Hull, *Options, Futures And Other Derivatives*. Prentice Hall, 2009.

[25] D. Watkins, *Fundamentals of matrix computations*. New York: Wiley, 1991.

[26] Y. Lin and D. Padua, "Analysis of Irregular Single-Indexed Arrays and its Applications in Compiler Optimizations," in *Procs. Int. Conf. on Compiler Construction*, 2000, pp. 202–218.

[27] C. E. Oancea and L. Rauchwerger, "Scalable Conditional Induction Variable (CIV) Analysis," in *Procs. Int. Symp. Code Generation and Optimization (CGO)*, 2015.

[28] Basel Committee on Banking Supervision, Bank for International Settlements, "Basel III: A global regulatory framework for more resilient banks and banking systems," December 2010.

[29] D. Brigo and F. Mercurio, *Interest Rate Models - Theory and Practice: With Smile, Inflation and Credit (Springer Finance)*, 2nd ed. Springer, Aug. 2006.

[30] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec. 1997.

[31] R. P. Brent, *Algorithms for Minimization without Derivatives*. Prentice-Hall, 1973.

[32] N. M. Steen, G. D. Byrne, and E. M. Gelbard, "Gaussian quadratures for the integrals $\int_0^\infty \exp(-x^2)f(x)dx$ and $\int_0^b \exp(-x^2)f(x)dx$," *Math. Comp.*, vol. 23, pp. 661–671, 1969.

[33] M. Joshi, "Graphical Asian Options," *Wilmott J.*, vol. 2, no. 2, pp. 97–107, 2010.

[34] A. Lee, C. Yau, M. Giles, A. Doucet, and C. Holmes, "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods," *J. Comp. Graph. Stat*, vol. 19, no. 4, pp. 769–789, 2010.

[35] F. Nord and E. Laure, "Monte Carlo Option Pricing with Graphics Processing Units," in *Int. Conf. ParCo*, 2011.

[36] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Fransisco, CA, 2001.

[37] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *Int. Journal of Par. Prog*, vol. 20(1), pp. 23–54, 1991.

[38] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time Composition of Run-time Data and Iteration Reorderings," in *Procs. Int. Conf. Prog. Lang. Design and Implem. (PLDI)*. ACM, 2003, pp. 91–102.

[39] C. E. Oancea, A. Mycroft, and S. M. Watt, "A New Approach to Parallelising Tracing Algorithms," in *Procs. Int. Symp. on Memory Management (ISMM)*. ACM, 2009, pp. 10–19.

[40] F. C. Gieseke, J. Heinermann, C. E. Oancea, and C. Igel, "Buffer k-d trees: processing massive nearest neighbor queries on GPUs," in *Int. Conf. on Machine Learning (ICML)*, 2014.

[41] B. Armstrong and R. Eigenmann, "Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries," in *Int. Conf. Parallel Proc. (ICPP)*, 2008, pp. 279–286.

[42] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops," in *Int. Par. and Distr. Processing Symp. (PDPS)*, 2002, pp. 20–29.

[43] C. E. Oancea and A. Mycroft, "Set-Congruence Dynamic Analysis for Software Thread-Level Speculation," in *Procs. Langs. Comp. Parallel Computing (LCPC)*, 2008, pp. 156–171.

[44] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*, 2008, pp. 73–82.

[45] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems Jade Edition*, W.-m. W. Hwu, Ed. San Francisco, CA, USA: Morgan Kaufmann, 2011, ch. 26.

[46] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a High-Level Language for GPUs," in *Int. Conf. Prg. Lang. Design and Implem. (PLDI)*, 2012, pp. 1–12.

[47] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, "CUDA-Lite: Reducing GPU Programming Complexity," in *Int. Work. Lang. and Compilers for Par. Computing (LCPC)*, 2008, pp. 1–15.

[48] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Int. Conf. Prog. Lang. Design and Implem. (PLDI)*, 2010, pp. 86–97.

[49] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static Compilation Analysis for Host-Accelerator Communication Optimization," in *Int. Work. Lang. and Compilers for Par. Computing (LCPC)*, 2011.

[50] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs." in *Int. Conf. on Compiler Construction (CC)*, 2010, pp. 244–263.

[51] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization," in *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*, 2009, pp. 101–110.

[52] G. Mainland and G. Morrisett, "Nikola: Embedding Compiled GPU Functions in Haskell," in *Proceedings of the 3rd ACM International Symposium on Haskell*, 2010, pp. 67–78.

[53] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell Array Codes with Multicore GPUs," in *International Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP'11, 2011, pp. 3–14.

[54] K. Claessen, M. Sheeran, and B. J. Svensson, "Expressive Array Constructs in an Embedded GPU Kernel Programming Language," in *International Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP'12, 2012, pp. 21–30.

[55] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution," *Int. Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, 2006.

[56] C. Grelck, "Shared memory multiprocessor support for functional array processing in SAC," *Journal of Functional Programming (JFP)*, vol. 15, no. 3, pp. 353–401, 2005.

[57] J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the GPU programming barrier with the auto-parallelising SAC compiler," in *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*. ACM, 2011, pp. 15–24.

[58] K. E. Iverson, *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[59] C. Grelck and S.-B. Scholz, "Accelerating APL programs with SAC," in *Proceedings of the Conference on APL '99: On Track to the 21st Century*, ser. APL'99. ACM, 1999, pp. 50–57.

[60] G. Hains and L. M. R. Mullin, "Parallel functional programming with arrays," *The Computer Journal*, vol. 36, no. 3, pp. 238–245, 1993.

[61] M. Elsman and M. Dybdal, "Compiling a Subset of APL Into a Typed Intermediate Language," in *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[62] C. B. Jay, "Programming in fish," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 307–315, 1999.

[63] G. Blelloch, "Programming Parallel Algorithms," *Communications of the ACM (CACM)*, vol. 39, no. 3, pp. 85–97, 1996.

[64] L. Bergstrom and J. Reppy, "Nested data-parallelism on the GPU," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, Sep. 2012, pp. 247–258.

[65] F. M. Madsen and A. Filinski, "Towards a streaming model for nested data parallelism," in *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, September 2013.

[66] Standard Performance Evaluation Corporation, "The SPEC ACCEL Benchmark," December 2014, Benchmark available from https://www.spec.org/accel.