FinPar: A Parallel Financial Benchmark

CHRISTIAN ANDREETTA, Nordea Capital Markets
VIVIEN BÉGOT, LexiFi
JOST BERTHOLD, MARTIN ELSMAN, FRITZ HENGLEIN, TROELS HENRIKSEN, MAJ-BRITT NORDFANG, and COSMIN E. OANCEA, University of Copenhagen

Commodity many-core hardware is now mainstream, but parallel programming models are still lagging behind in efficiently utilizing the application parallelism. There are (at least) two principal reasons for this. First, real-world programs often take the form of a deeply nested composition of parallel operators, but mapping the available parallelism to the hardware requires a set of transformations that are tedious to do by hand and beyond the capability of the common user. Second, the best optimization strategy, such as what to parallelize and what to efficiently sequentialize, is often sensitive to the input dataset and therefore requires multiple code versions that are optimized differently, which also raises maintainability problems.

This article presents three array-based applications from the financial domain that are suitable for GPGPU execution. Common benchmark-design practice has been to provide the same code for the sequential and parallel versions that are optimized for only one class of datasets. In comparison, we document (1) all available parallelism via nested `map-reduce` functional combinators, in a simple Haskell implementation that closely resembles the original code structure, (2) the invariants and code transformations that govern the main trade-offs of a data-sensitive optimization space, and (3) report target CPU and multiversion GPGPU code together with an evaluation that demonstrates optimization trade-offs and other difficulties. We believe that 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.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Source code generation**; **Software performance;**

Additional Key Words and Phrases: Data-parallel functional language, fusion, fission, strength reduction

ACM Reference Format:
Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A parallel financial benchmark. ACM Trans. Archit. Code Optim. 13, 2, Article 18 (June 2016), 27 pages.
DOI: http://dx.doi.org/10.1145/2898354

This work was partially supported by the Danish Council for Strategic Research under contract 10-092299 (HIPERFIT: http://hiperfit.dk). C. Andreetta contributed to this article in his personal capacity; the views expressed are his own and do not necessarily represent the views of Nordea.

Authors' addresses: C. Andreetta, Nordea Capital Markets, Strandgade 3, 1401 Copenhagen, Denmark; email: christian.andreetta@nordea.com; V. Bégot, 892 Rue Yves Kermen, 92100 Boulogne-Billancourt, France; email: vivien.begot@lexifi.com; J. Berthold, Commonwealth Bank of Australia, 1 Harbour St, 2000 Sydney, NSW; email: jberthold@acm.org; M.-B. Nordfang, Department of Mathematical Sciences, University of Copenhagen, Nørre Campus, Universitetsparken 5, DK-2100 Copenhagen, Denmark; email: mbnordfang@math.ku.dk; M. Elsman, F. Henglein, T. Henriksen, and C. E. Oancea, Department of Computer Science, University of Copenhagen, Nørre Campus, Universitetsparken 5, DK-2100 Copenhagen, Denmark; emails: {mael, henglein, athas, cosmin.oancea}@diku.dk.

DOI: http://dx.doi.org/10.1145/2898354

1. INTRODUCTION

With the mainstream emergence of many-core architectures, such as GPGPUs, massive parallelism has become a focus area of industrial application development. However, parallel programming models are lagging behind the advance in hardware: parallelism extraction and optimization is still tedious and often requires specialized knowledge.

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 such as 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 [Blelloch et al. 1994] (NESL), polyhedral frameworks [Pouchet et al. 2011] (C), or interprocedural summarization of array subscripts [Hall et al. 2005] (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 [Reinders 2007; Nguyen et al. 2014; Giles et al. 2011; Chakravarty et al. 2011].

This article presents a benchmark suite for optimizing compilers and parallel DSLS that comprises three large-scale modeling components of a financial engine: a pricing engine for financial contracts, a local volatility calibration method, and an interest rate calibration method based on observed swaption prices. (The results of the calibration methods in principle can be used for pricing.)

Our benchmark provides sequential (original) source code, ranging from hundreds to several thousands of lines of compact code, and different parallel versions for multicore and GPGPU execution. For example, the sequential code can be used to test autoparallelization solutions, with the parallel versions providing the comparison baseline.

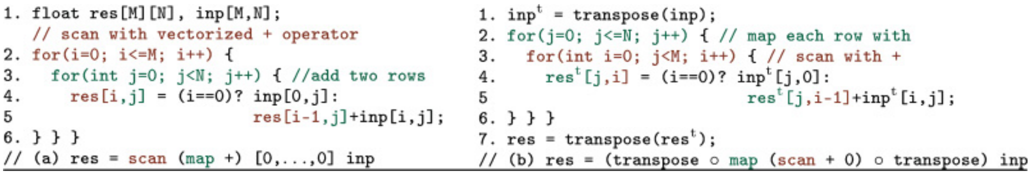
Although the initial motivation has been to efficiently parallelize the original code base (C and OCaml), the journey has quickly become more important than the destination. Since the code exhibits deeply nested parallelism in which the optimal parallelization strategy is sensitive to the input dataset, the focus has shifted toward (1) fully expressing all parallelism and algorithmic trade-offs with a set of generic data-parallel array operators and (2) studying how a compiler can exploit the compositional algebra of such operators to derive systematically implementations that are efficient for all datasets[1] (i.e., efficient optimization of the common case but within work-depth asymptotic guarantees). Work is in progress to integrate these findings into Futhark, a pure functional language and its compiler infrastructure [Henriksen 2014; Henriksen and Oancea 2013, 2014; Henriksen et al. 2014].

1.1. Main Contributions: "Why Yet Another Benchmark?"

Our work differs in several important ways from common benchmark practice.

First, if many-core architectures are to follow Moore's law, the programming model should assume infinite hardware parallelism, whereas efficient sequentialization of the excess parallelism should be just an optimization, albeit an important one. In this sense, we provide a pure functional Haskell implementation that retains the original program structure and specifies all available parallelism in terms of nested `map`, `reduce`, `scan`, and `filter` operations on lists (runtime is irrelevant). Furthermore, the higher-level semantics of such parallel basic blocks allows code transformations to be reasoned about in terms of simple but powerful rewrite rules rather than in terms

[1] Supporting nested parallelism is important, because although companies are eager to reap the benefits of many-core architectures, they are unwilling to rewrite their (sequential) code base more than once, and only if the resulting code structure still resembles the original algorithm. However, static extraction and optimization of parallelism for GPGPU requires a set of transformations (e.g., fusion, fission, loop interchange) that are tedious, nontrivial, and result in an unmaintainable code base.



```
float res[M][N], inp[M,N];
// scan with vectorized + operator
for(i=0; i<=M; i++) {
  for(int j=0; j<N; j++) { //add two rows
    res[i,j] = (i==0)? inp[0,j]:
                             res[i-1,j]+inp[i,j];
} } }
// (a) res = scan (map +) [0,...,0] inp
```

```
inp^t = transpose(inp);
for(j=0; j<=N; j++) { // map each row with
  for(int i=0; j<M; i++) { // scan with +
    res^t[j,i] = (i==0)? inp^t[j,0]:
                           res^t[j,i-1]+inp^t[i,j];
} } }
res = transpose(res^t);
// (b) res = (transpose ∘ map (scan + 0) ∘ transpose) inp
```

Fig. 1. Imperative code for scan with vectorized + (a) and segmented scan with + (scan each row) (b).

of tedious and conservative dependence analysis of array indices. As an important example, we have found that (segmented) scan, a well-known basic block of parallel programming [Blelloch 1989], plays a predominant role in our benchmark, albeit it appears rarely (if at all) in other parallel benchmarks.

One possible reason could be that scan instances are difficult to recognize and optimize in imperative code. Figure 1(a) shows a scan with the vectorized addition operator—that is, each column of res will contain the prefix sums of each column in inp. Even if recognized, such a scan with a vectorized operator is unsuitable for GPGPU execution. However, if recognized, the user or compiler can reason equationally about the use of the high-level transformation, shown in Figure 1(b), which moves the scan inward; if the input array is transposed, then the prefix sums can be computed row-wise and the result transposed back. This transformation would allow for efficiently exploiting the parallelism of both loops (map and scan), which would correspond to a segmented scan with a scalar addition operator (segments are rows).

Second, to a large extent, current solutions employ a "one size fits all" parallelization strategy that results in one target program for all datasets. For example, in OpenCL, the user explicitly specifies which operations are executed in parallel and which are sequential. Moreover, in a purely functional context, the flattening transformation [Blelloch 1996] offers work-depth asymptotic guarantees for the parallelized program but does not optimize memory usage, communication, or locality of reference. In addition, although imperative solutions typically optimize the common case (maps and locality of reference), they do so without providing asymptotic guarantees.

In contrast, we take the perspective that the rich trade-off space revealed by the functional specification is a generator of low-level implementations that exhibit very different intrinsic behavior, such as the degree of parallelism, the number of executed instructions, and their dynamic mix. In this sense, we provide documentation of the important trade-offs and program transformations that exploit them, together with realistic datasets that demonstrate these trade-offs. 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 (or anywhere in between, i.e., moderate flattening). For example, local volatility calibration uses the tridiagonal solver (TRIDAG), which appears as a fully dependent loop but can can be rewritten into scans in which the associative operators are linear-function composition and 2×2 matrix multiplication, respectively. These scan instances are expensive to parallelize, yielding significant instructional overhead and $\log n$ depth, but are necessary for two out of three datasets to fully utilize hardware parallelism. Another example is the strength reduction invariant in the computation of Sobol numbers: a computationally expensive independent formula that requires map parallelism versus a cheaper recurrent formula that requires scan.

Finally, although several components of the proposed applications may seem to overlap with current benchmarks, a closer look reveals significant algorithmic differences and challenges. For example, Black-Scholes is typically applied to price simple European call options, which are defined in terms of one asset measured at one date and whose implementation corresponds to a flat map on scalar operations [Chakravarty et al. 2011]. Our option pricing engine generalizes to more complex contracts, which are defined in terms of multiple assets measured at multiple dates, and it requires correlation of the sampled prices across both the asset and date dimensions. In terms of implementation, this added complexity translates to deeply nested map-scan parallelism (e.g., see Figure 6(b)). A second example is the GPU parallelization of the tridiagonal solver (TRIDAG). Related solutions [Kim et al. 2011; Egloff 2011] use cyclic reduction [Hockney 1965], which is a two-way elimination method for tridiagonal matrices and is very different, mathematically, from the sequential solution. It requires fairly sophisticated implementation techniques to resolve interthread dependencies in a scalable manner. In comparison, our parallel implementation can be derived systematically by compiler/language techniques from the sequential one by pattern matching and replacing the two well-known recurrences of TRIDAG with scans [Blelloch 1990], which also guarantees scalability. Furthermore, the TRIDAG-based parallelization of local volatility calibration follows and answers a direction for research proposed by Egloff [2011]. In summary, the principal contributions of this article are as follows:

—A complete description of the available parallelism in three big-compute applications.
—A detailed exploration of the main trade-offs of a dataset-sensitive optimization space that generates low-level implementations with different intrinsic behavior. Trade-off examples include moderate flattening, fusion versus fission, and strength reduction.
—Language constructs that can expose the optimization trade-offs to the compiler infrastructure, and code transformations that can take advantage of them.
—Baseline multicore/GPGPU code and an evaluation that demonstrates the trade-offs in terms of both parallel runtimes and other characteristics, such as dynamic instruction mixes, cache miss ratio, global memory footprint, and bandwidth utilization.

2. PRELIMINARIES

This section briefly presents the financial motivation for studying the three applications, the functional notation that we used to describe the available application parallelism and code transformations, and the experimental methodology.

2.1. Motivation for High-Performance Financial Computations

The financial system is facing fundamental computational challenges, led by an increase in complexity, interconnectedness, and speed of interaction between participants. Considering that financial institutions relocate capital across economic sectors, and are thus instrumental in providing stable economical growth, should a large institution face liquidity shortage, a set of cascade effects may negatively impact the whole system. The impact of capital allocation across a large number of forecasted scenarios is estimated via large-scale simulations, which present a compelling and challenging application for commodity many-core hardware (e.g., GPGPUS), albeit they transcend the domain of embarrassingly parallel computing. For example, Monte Carlo simulations, originally developed to investigate the stochastic behavior of physical systems in complex, multidimensional spaces, have emerged as a tool of choice in critical financial applications, such as risk modeling and contract pricing. In these simulations, gains and risks can be described by means of a probabilistic formulation of possible market scenarios, estimated and aggregated with a Monte Carlo method, and evaluated at present time by a discount function. This article presents three components used in practice to implement such a task.

Section 3 presents a pricing engine for a set of vanilla and exotic options in scenarios with known volatility. Section 4 presents a method for local volatility calibration, in which market volatility is modeled as a parameter of the option price. The volatility is calibrated by solving a system of continuous partial differential equations (PDEs)

Types: τ ::= `bool` | `char` | `int` | `real` | $(\tau_1, \ldots, \tau_n)$ | $[\tau, n]$// basic types, tuples and array (of outer size n)

Notation: n, m `integers,` a, b `arrays,` f, g `functions,` $\oplus$ `binary associative operator,` $\#$ `array concatenation.`

Types and Semantics of Array Constructor and Second-Order Combinators (SOAC)

zip $:: ([\alpha_1, n], \ldots, [\alpha_m, n]) \rightarrow [(\alpha_1, \ldots, \alpha_m), n]$	**zip**$(a_1, \ldots, a_m) \equiv [(a_1[0], \ldots, a_m[0]), \ldots]$
unzip $:: [(\alpha_1, \ldots, \alpha_m), n] \rightarrow ([\alpha_1, n], \ldots, [\alpha_m, n])$	**unzip** $\equiv$ **zip**$^{-1}$
replicate $:: (\texttt{int}\ n, \alpha) \rightarrow [\alpha, n]$	**replicate**$(n, a) \equiv [a, \ldots, a]$ // array of outer size n
iota $:: \texttt{int}\ n \rightarrow [\texttt{int}, n]$	**iota**$(n) \equiv [0, \ldots, n-1]$
map $:: ((\alpha \rightarrow \beta),\ [\alpha, n]) \rightarrow [\beta, n]$	**map**$(f, a) \equiv [f(a[0]), f(a[1]), \ldots]$
filter $:: \exists m \leq n.((\alpha \rightarrow \texttt{bool}),\ [\alpha, n]) \rightarrow [\alpha, m]$	**filter**$(f, a) \equiv [a[i] \mid f(a[i]) = \texttt{True}]$
reduce $:: (((\alpha, \alpha) \rightarrow \alpha),\ \alpha,\ [\alpha, n]) \rightarrow \alpha$	**reduce**$(\oplus, e, a) \equiv e \oplus a[0] \oplus a[1] \ldots \oplus a[n-1]$
scan $:: (((\alpha, \alpha) \rightarrow \alpha),\ \alpha,\ [\alpha, n]) \rightarrow [\alpha, n]$	**scan**$(\oplus, e, a) \equiv [e \oplus a[0], e \oplus a[0] \oplus a[1], \ldots]$
streamPar $:: ((\forall m.[\beta, m] \rightarrow [\gamma, m]),\ [\beta, n]) \rightarrow [\gamma, n]$	**streamPar** $(g,\ a) \equiv g(a) \equiv g(a_1)\#\ldots\#g(a_s)$ $\forall$ partition of $a \equiv (a_1\#\ldots\#a_n)$
streamRed $:: (\ ((\alpha,\ \alpha) \rightarrow \alpha),\ (\forall m.(\alpha, [\beta, m]) \rightarrow \alpha)$ $,\ \alpha,\ [\beta, n]\ ) \rightarrow \alpha$	**streamRed** $(\oplus, g, e, a) \equiv e \oplus b_1 \oplus \ldots \oplus b_s$ $\forall a \equiv (a_1\#\ldots\#a_s)$ and $b_i = g(e, a_i)$
streamSeq $:: (\ (\forall m.(\alpha,\ [\beta, m]) \rightarrow (\alpha,\ [\gamma, m]))$ $,\ \alpha,\ [\beta, n]\ ) \rightarrow (\alpha,\ [\gamma, n])$	**streamSeq** $(g,\ e_0,\ a) \equiv (e_s,\ b_1\#\ldots\#b_s)$ $\forall a \equiv (a_1\#\ldots\#a_s)$ and $(e_i, b_i) = g(e_{i-1}, a_i)$

Fig. 2. Types and semantics of array constructors and second-order combinators.

using Crank-Nicolson's finite differences method [Munk 2007]. Section 5 presents a calibration of the parameters of an interest rate model based on a set of available swaption prices. The interest rate model can be used to value financial products, such as interest rate swaps.

2.2. Functional Language Notation

This work uses a (pure) functional notation that resembles ML [Milner et al. 1997]. In particular, it assumes strict evaluation and supports let bindings for local variables. However, unlike ML, in our notation, user-defined functions are monomorphic and their return and parameter types are specified explicitly. Figure 2 presents the types and semantics of (some of) the built-in polymorphic, second-order functions (SOACs) that can be used to construct and combine arrays.

Types include `char`, `bool`, `int`, `real`, tuples, and multidimensional regular arrays (i.e., arrays for which all subarray elements have identical shapes). Array shapes can be optionally specified in the declaration of functions (e.g., `[[int],n]` denotes a matrix with `n` rows). The operations `zip` and `unzip` move between the array-of-tuple and tuple-of-array representations, and `iota` and `replicate` build the iteration-space and same-value array, respectively. The operations `map`, `filter`, `reduce`, and `scan` are the parallel array operators of the Bird-Meertens formalism [Bird 1987] and are used for mapping over an array with a function, filtering the elements of an array by a predicate, and reducing and computing all prefix sums of an array's elements by a binary-associative operator. Anonymous (or curried) functions are syntactically permitted only as SOAC function arguments. For instance, the code `let t =... in map (fn int (int x)=>x+t, iota(t))` builds the result array by adding `t` to each element of `[0...t-1]`, where `t` is defined in the outer context. Finally, three types of stream operations are useful to express high-level invariants, such as strength reduction, and to complete the compositional SOAC algebra, which includes laws for fusion and fission:

- The `streamPar` and `streamRed` operations apply the input function g to each element of an arbitrary partitioning of the input array a, and concatenates or reduces the resulting arrays or values, respectively. Both streams allow chunks to be processed in parallel, but the user must ensure that any partitioning gives the same result.

—The `streamSeq` operation features sequential chunk execution; the result e_i of processing chunk a_i becomes the (accumulator) input for processing the next chunk $i + 1$.

Notice that the SOACS have implicit data-parallel semantics; they not only declare parallelism (as in OPENMP) but also guarantee data-race free programs.

2.3. Experimental Methodology and Code Availability

The parallel and sequential runtimes are averaged across 20 runs, and uncertainties are reported with 95% confidence. The reported runtimes include all overheads except (1) reading the input data and validating and writing the result to file, and (2) GPU context creation and build time. (The memory transfer time between host and device is accounted for but is negligible in most cases because all computation is performed on GPGPU and the memory size of the input dataset and result are small.) The multicore and GPU code is implemented in OPENMP and OPENCL, and is compiled with GCC 4.8.4 (`-fopenmp -O3`) and CUDA 6.0, respectively. The code is run on an Intel system, using 16 Xeon cores, model `E5-2650 v2`, running at 2.60GHz, each supporting 2-way hyperthreading, and with a 32`KB`, 8-way associative private `L1` instruction and data caches, and a 20`MB`, 20-way associative shared last-level cache. The GPGPU is a GeForce GTX 780 Ti NVIDIA, with 3Gbytes of global memory, 2,880 cores running at 1.08GHz, and 1.5Mbytes of L2 cache. We use `pine` and `valgrind` to compute the instruction mixes and the cache utilization of the multicore execution.

The benchmark is publicly available at `https://github.com/HIPERFIT/finpar`. Binaries for the differently transformed versions of the same program can be easily obtained (via `make` scripts), because versions either have their own (separate) `C/OpenCL` implementation, such as the `ALL` and `OUT` versions of volatility calibration, or can be simply derived by macro definitions, such as with the `OpenCL` version of option pricing.[2]

3. OPTION PRICING BENCHMARK

The presentation is organized as follows. Section 3.1 presents the main components of option pricing and shows that the benchmark translates directly to a nested map-reduce function composition, which expresses well the algorithmic structure and the available parallelism. Sections 3.2 and 3.3 study some of the high-level invariants and trade-offs that govern the optimization space, such as strength reduction and fusion.

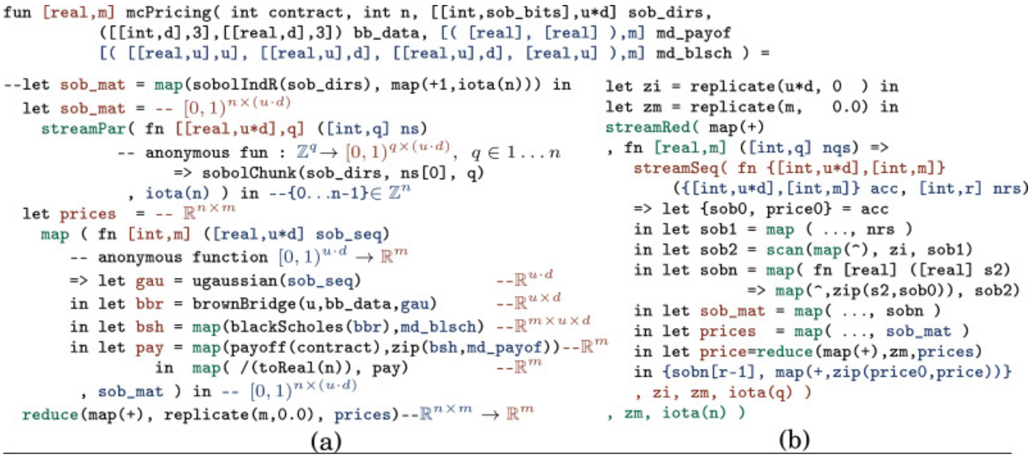
Section 3.4 compares against an imperative setting: it identifies several key imperative code patterns, such as `scans`, that would seriously hinder parallelism detection, and it motivates the need for supporting loops with in-place updates and for performing lower-level optimizations, such as memory coalescing. Section 3.5 discusses at a high level how these optimizations can be supported in a functional language.

Finally, Section 3.6 presents an empirical evaluation that demonstrates the impact of optimizations such as fusion, strength reduction, and memory coalescing, and shows that "one size does *not* fit all": the dataset-sensitive optimization space leads to several low-level implementations exhibiting very different characteristics in terms of parallel runtime, dynamic instruction mixes, memory footprint, and so forth.

3.1. Functional Basic Blocks of the Pricing Engine

Option contracts are among the most commonly exchanged instruments between financial actors. They are formulated in terms of a set of (1) trigger conditions on

[2] Currently, options such as fusion/fission, memory coalescing, and Sobol strength reduction can be (de)selected in file `OptionPricing/includeC/Optimizations.h`, but in general, please follow the `README.md` instructions.



```
fun [real,m] mcPricing( int contract, int n, [[int,sob_bits],u*d] sob_dirs,
          ([[int,d],3],[[real,d],3]) bb_data, [( [real], [real] ),m] md_payof
          [( [[real,u],u], [[real,u],d], [[real,u],d], [real,u] ),m] md_blsch ) =
```

```
--let sob_mat = map(sobolIndR(sob_dirs), map(+1,iota(n))) in
  let sob_mat = -- [0,1)^{n×(u·d)}
    streamPar( fn [[real,u*d],q] ([int,q] ns)
             -- anonymous fun : ℤ^q → [0,1)^{q×(u·d)},  q ∈ 1...n
                 => sobolChunk(sob_dirs, ns[0], q)
           , iota(n) ) in --{0...n-1}∈ ℤ^n
  let prices  = -- ℝ^{n×m}
    map ( fn [int,m] ([real,u*d] sob_seq)
       -- anonymous function [0,1)^{u·d} → ℝ^m
       => let gau = ugaussian(sob_seq)              --ℝ^{u·d}
       in let bbr = brownBridge(u,bb_data,gau)      --ℝ^{u×d}
       in let bsh = map(blackScholes(bbr),md_blsch) --ℝ^{m×u×d}
       in let pay = map(payoff(contract),zip(bsh,md_payof))--ℝ^m
               in  map( /(toReal(n)), pay)          --ℝ^m
     , sob_mat ) in -- [0,1)^{n×(u·d)}
  reduce(map(+), replicate(m,0.0), prices)--ℝ^{n×m} → ℝ^m
```

(a)

```
let zi = replicate(u*d, 0  ) in
let zm = replicate(m,   0.0) in
streamRed( map(+)
, fn [real,m] ([int,q] nqs) =>
    streamSeq( fn {[int,u*d],[int,m]}
          ({[int,u*d],[int,m]} acc, [int,r] nrs)
    => let {sob0, price0} = acc
    in let sob1 = map ( ..., nrs )
    in let sob2 = scan(map(^), zi, sob1)
    in let sobn = map( fn [real] ([real] s2)
                    => map(^,zip(s2,sob0)), sob2)
    in let sob_mat = map( ..., sobn )
    in let prices  = map( ..., sob_mat )
    in let price=reduce(map(+),zm,prices)
    in {sobn[r-1], map(+,zip(price0,price))}
    , zi, zm, iota(q) )
, zm, iota(n) )
```

(b)

Fig. 3. Pricing engine: functional basic blocks and types (a) and code after fusion (b).

market events; (2) mathematical dependencies over a set of assets, named *underlyings* of the contract; and (3) trigger dates: the time at which 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 trigger dates. The parts of this component are described in more detail in the remainder of this section.

—A technique to efficiently estimate the expected payoff by aggregating over the stochastic exploration. This component uses the quasirandom Monte Carlo method [Glasserman 2004], which, in simple terms, averages over a population of prices obtained in the previous step by regular, equidistant sampling.

The function `mcPricing` in Figure 3(a) shows a map-reduce implementation of the algorithm, the types of main components and the manner in which they are composed. Its first two arguments denote the number of the `contract` that is to be priced and the number of Monte Carlo iterations `n` to be used for pricing. In addition, integer variables `d`, `u`, and `m` denote the number of trigger dates, underlyings, and market scenarios, respectively, and are implicitly declared in the shape declaration of the other array parameters.[3] The result of `mcPricing` is 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 stochastic exploration proceeds by drawing samples from an equi-probable, homogeneous distribution. This step could have been expressed simply by (`map`ping) the independent-Sobol formula `sobolIndR`, which produces a pseudorandom sequence of size $u \cdot d$ when applied to an integer in `{1..n}`. Instead, this is commented (`--`) and replaced with a more efficient (and still parallel) implementation that uses `streamPar` to compute a chunk of Sobol sequences at a time (see Section 3.2).

[3] The other (array) parameters of `mcPricing` are invariant to the stochastic exploration and are used in various stages of the algorithm. For example, `sob_bits` denote the number of bits in the Sobol integer representation, `sob_dir_vcts` are Sobol's direction vectors, `bb_data` are parameters of the Brownian bridge, and `md_blsch` and `md_payof` are the parameters of `m` market scenarios, such as volatility, discount, and so on.

The remaining code is at the outermost level a `reduce` ∘ `map` composition applied to the array of `n` Sobol sequences `sob_mat`. The `map` corresponds to the rest of the stochastic exploration and results in a matrix of `prices`. The `reduce` implements the Monte Carlo aggregation by adding componentwise (`map(+)`) the `n` price vectors produced by the `map`. The neutral element is a vector of `m` zeros, and the result belongs to $\mathbb{R}^m$.

The implementation of the functional parameter of the (outermost) `map` is semantically the composition of four functions. First, the uniform samples are mapped by quantile-probability inversion [Wichura 1988] to normally distributed values, and they are later used to model the value of each underlying at the trigger dates. This mapping is performed by the function `ugaussian`, which has type $[0, 1)^{u \cdot d} \rightarrow \mathbb{R}^{u \cdot d}$.

Second, since the market is assumed to present good liquidity and no discontinuities, the underlyings are independently modeled as geometric Brownian motions [Black and Scholes 1973]: continuous stochastic processes whose increments follow a log-normal distribution. This step corresponds to the `brownBridge` call, which correlates the input samples along the date dimension, independently for each underlying, to impose the properties of the stochastic processes [Hull 2009] also on nonobserved dates. It follows that the input vectors and the result are reshaped to `u` × `d` matrices in which correlation is performed in each row.

Third, the inner `map`s estimate the contract price for each of the `m` market scenarios:

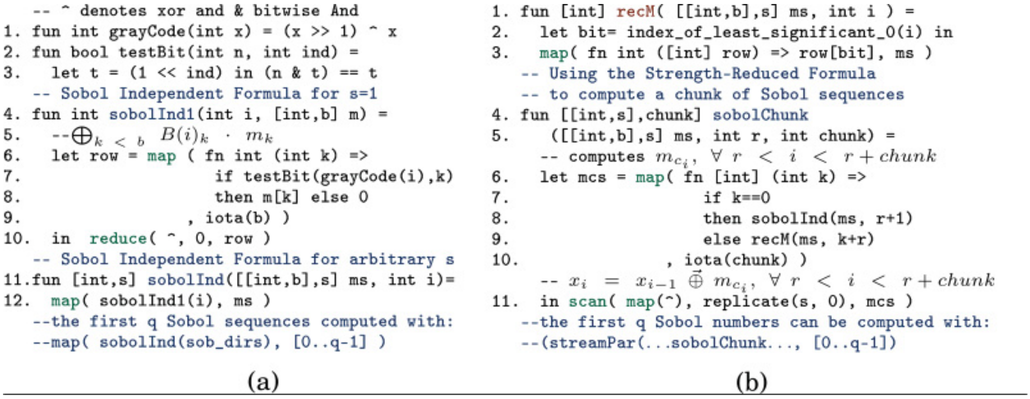
(1) To express the expected correlation among underlyings, `blackScholes` scales once again the input samples via Cholesky composition by means of a positive-definite correlation matrix [Watkins 1991], which is part of `md_blsch`.
(2) The obtained samples now mimic a (particular) market scenario and are provided as input to the `payoff` function that calculates the future gain from the contract in the current sample and the value of the aggregated future payoff at present time via a suitable market discount model [Hull 2009].
(3) The obtained prices are divided by `n` such that the average will result by summation.

We conclude with two remarks. First, having precise array shape information is fundamental for achieving efficient parallelization, as it enables, for example, hoisting allocations outside recurrences and loop distribution (which require array expansion). In this sense, previous work reported a slicing technique that infers (optimized) code that computes precise shapes at all array creation points [Henriksen et al. 2014]. Second, the functional notation borrows the expressiveness of Bird-Marteen's formalism for specifying parallelism and high-level invariants, which are discussed next.

3.2. Sobol Independent Versus Strength Reduction Formula Trade-Off

A Sobol sequence [Bratley and Fox 1988] is an example of a quasirandom 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 (of discrepancy $O(\frac{\log^s n}{n})$). The Sobol algorithm for $s = 1$ starts by computing a number of direction vectors m_k, where each m_k is a positive integer and there are as many ks as bits in the integer representation (`sob_bits`). This step is not explained here because this computation is not on the critical path (i.e., m_k are computed once and used many times).

The ith 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 kth bit of the canonical bit representation of the positive integer i, and $\oplus$ denotes the exclusive-or operator. In the preceding 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.



```
-- ^ denotes xor and & bitwise And
fun int grayCode(int x) = (x >> 1) ^ x
fun bool testBit(int n, int ind) =
  let t = (1 << ind) in (n & t) == t
-- Sobol Independent Formula for s=1
fun int sobolInd1(int i, [int,b] m) =
  --⊕_{k < b} B(i)_k · m_k
  let row = map ( fn int (int k) =>
                      if testBit(grayCode(i),k)
                      then m[k] else 0
                    , iota(b) )
  in  reduce( ^, 0, row )
-- Sobol Independent Formula for arbitrary s
fun [int,s] sobolInd([[int,b],s] ms, int i)=
  map( sobolInd1(i), ms )
--the first q Sobol sequences computed with:
--map( sobolInd(sob_dirs), [0..q-1] )
```

(a)

```
fun [int] recM( [[int,b],s] ms, int i ) =
  let bit= index_of_least_significant_0(i) in
  map( fn int ([int] row) => row[bit], ms )
-- Using the Strength-Reduced Formula
-- to compute a chunk of Sobol sequences
fun [[int,s],chunk] sobolChunk
   ([[int,b],s] ms, int r, int chunk) =
  -- computes m_{c_i}, ∀ r < i < r + chunk
  let mcs = map( fn [int] (int k) =>
                    if k==0
                    then sobolInd(ms, r+1)
                    else recM(ms, k+r)
                 , iota(chunk) )
  -- x_i = x_{i-1} ⊕⃗ m_{c_i}, ∀ r < i < r + chunk
  in scan( map(^), replicate(s, 0), mcs )
--the first q Sobol numbers can be computed with:
--(streamPar(...sobolChunk..., [0..q-1])
```

(b)

Fig. 4. Computing Sobol sequences in Futhark by independent (a) and strength-reduced (b) formulas.

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

Figure 4(a) shows the sobolInd Futhark function that implements the Sobol independent formula by mapping the formula for $s = 1$, denoted sobolInd1, on each of the s direction vectors, denoted ms. Note that the implementation reveals two (extra) nested levels of parallelism of sizes s and sob_bits=30, which can be easily exploited.

Figure 4(b) shows the function sobolChunk, which uses the strength-reduced (recurrent) formula to compute a chunk of Sobol sequences by applying the independent formula to compute the first element of the chunk and amortizing this overhead by using the efficient recurrent formula for the rest of the chunk. The implementation is a scan ∘ map composition, in which the map computes the $\mathtt{m}_c$ contributions and the scan applies the recurrent formula to all elements in the chunk.

The trade-off refers to which formula to use for computing n consecutive Sobol numbers. The independent formula can simply be mapped, and hence it enables efficient parallelization of depth $O(1)$ but requires significantly more work than the recurrent formula. The latter is computationally cheaper but requires a scan with the vectorized xor operator, which exhibits less efficient parallelism of $O(\log n)$ depth.

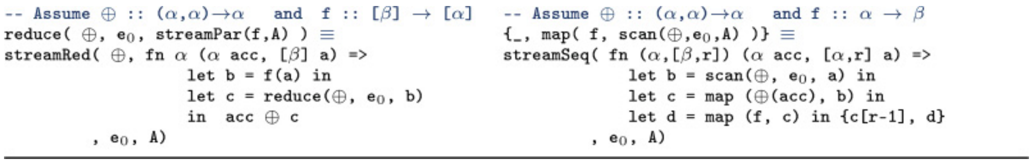
An elegant solution that combines advantages was already shown in Figure 3(a): streamPar is used to express the strong invariant that chunks of an arbitrary partitioning of the [0..n-1] array can be processed in parallel ($O(1)$ depth) via sobolChunk. Thus, the chunk size is a free parameter that can be instantiated by the compiler to the best-suited value, for example by generating two code versions that are discriminated based on the value of data-sensitive input n, the Monte-Carlo space size.

If hardware parallelism approaches n, then choosing chunk size 1 would essentially transform the streamPar call to a map with sobolInd, thus taking full advantage of the efficient outer parallelism and providing opportunities to exploit inner parallelism.

Otherwise, the excess parallelism can be efficiently sequentialized via the recurrent formula by using n/hwp as chunk size, where hwp accounts for hardware parallelism.

3.3. Fusion Versus Fission Trade-Off

The second high-level trade-off corresponds to two well-known invariants [Bird 1987]. Map-map fusion (fission) states that mapping the elements of an array with a function and then the result with another function is equivalent to mapping the original array



```
-- Assume ⊕ :: (α,α)→α   and  f :: [β] → [α]
reduce( ⊕, e0, streamPar(f,A) ) ≡
streamRed( ⊕, fn α (α acc, [β] a) =>
                   let b = f(a) in
                   let c = reduce(⊕, e0, b)
                   in  acc ⊕ c
         , e0, A)
```

```
-- Assume ⊕ :: (α,α)→α   and f :: α → β
{_, map( f, scan(⊕,e0,A) )} ≡
streamSeq( fn (α,[β,r]) (α acc, [α,r] a) =>
             let b = scan(⊕, e0, a) in
             let c = map (⊕(acc), b) in
             let d = map (f, c) in {c[r-1], d}
         , e0, A)
```

Fig. 5. Fusion rules: `reduce ∘ streamPar ⇒ streamRed (a)` and `map ∘ scan ⇒ streamSeq (b)`.

with the composition of the two functions:

$$(\texttt{map g}) \circ (\texttt{map f}) \equiv \texttt{map}(g \circ f). \tag{1}$$

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-processor arrays of equal sizes, on which each processor performs the original computation sequentially, and finally, the local results are reduced in parallel:

$$(\texttt{red}\ \oplus\ e) \circ (\texttt{map f}) \equiv (\texttt{red}\ \oplus\ e) \circ (\texttt{map}\ ((\texttt{red}\ \oplus\ e) \circ (\texttt{map f}))) \circ \texttt{dist}_p. \tag{2}$$

For example, with the option pricing code in Figure 3(a)—but in which Sobol sequences are computed by mapping the independent formula—the two outermost maps can be fused by Equation (1) and the result can be fused with the reduce that performs the Monte Carlo aggregation by Equation (2). However, the direction in which these invariants should be applied to maximize performance is sensitive to the input dataset.

If the memory footprint of a *fused* iteration, 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 (1) slow (global) memory accesses are eliminated from the critical path, and hence (2) the execution behavior becomes compute rather than memory bound, and (3) the memory consumption is reduced asymptotically (i.e., 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). This strategy (1) allows one to exploit more parallelism, such as the inner maps of degree m in Figure 3(a) and the inner parallelism of each component, and (2) leads to simpler kernels that exhibit less register pressure (and better hardware utilization).

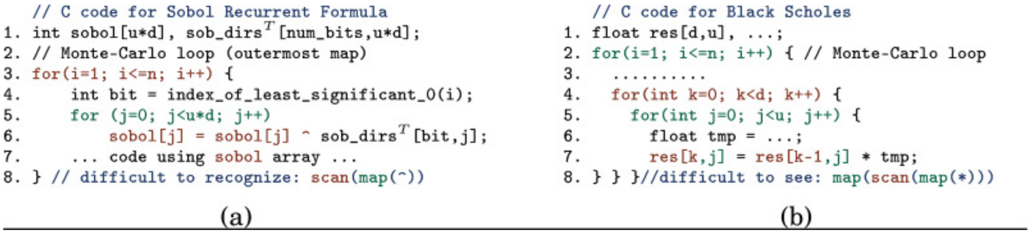
Finally, Figure 3(b) shows that even when option pricing uses the efficient computation of Sobol sequences (i.e., the parallel stream with sobolChunk), the code is still fused into a perfect nest formed by streamRed and streamSeq at outer and inner levels, respectively. The parallel stream (streamRed) is obtained as the result of the outermost SOAC composition reduce ∘ map ∘ streamPar: the map is fused first with the streamPar resulting in a new streamPar SOAC, which is fused then with the reduce resulting in a streamRed according to the fusion rule shown in Figure 5(a).

The outer-level fusion creates opportunities for fusing at an inner level: the sequential stream (streamSeq) appears due to the use of scan in sobolChunk, which is fused with the subsequent map and reduce operators. For example, the rule for fusing a scan with a map into a streamSeq is shown in Figure 5(b).

In essence, the fused program enables efficient sequentialization of the code: (1) the outer streamRed enables the use of the cheaper Sobol recurrent formula and of a chunk size that is tunable to the input dataset, and (2) the inner streamSeq, if its chunk size is chosen 1, results in a loop whose memory footprint is proportional with $u \cdot d \cdot m$ and does not depend on the chunk size of the outer parallel stream streamRed (or n).

3.4. Comparison with the Imperative Setting

Figure 6 shows two of the many forms in which the scan primitive hides in imperative dependent loops. The code on the left side, corresponding to the Sobol strength-reduced



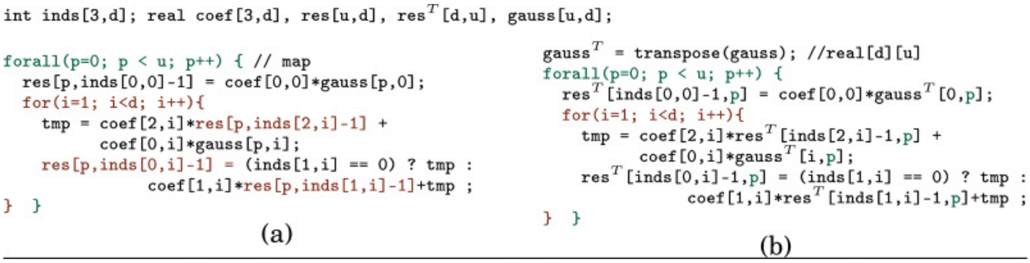
```
   // C code for Sobol Recurrent Formula
1. int sobol[u*d], sob_dirs^T[num_bits,u*d];
2. // Monte-Carlo loop (outermost map)
3. for(i=1; i<=n; i++) {
4.     int bit = index_of_least_significant_0(i);
5.     for (j=0; j<u*d; j++)
6.         sobol[j] = sobol[j] ^ sob_dirs^T[bit,j];
7.     ... code using sobol array ...
8. } // difficult to recognize: scan(map(^))
```

(a)

```
   // C code for Black Scholes
1. float res[d,u], ...;
2. for(i=1; i<=n; i++) { // Monte-Carlo loop
3.   ..........
4.   for(int k=0; k<d; k++) {
5.     for(int j=0; j<u; j++) {
6.       float tmp = ...;
7.       res[k,j] = res[k-1,j] * tmp;
8. } } }//difficult to see: map(scan(map(*)))
```

(b)

Fig. 6. Two original code snippets hiding scan(map) operators.



```
int inds[3,d]; real coef[3,d], res[u,d], res^T[d,u], gauss[u,d];

forall(p=0; p < u; p++) { // map
  res[p,inds[0,0]-1] = coef[0,0]*gauss[p,0];
  for(i=1; i<d; i++){
    tmp = coef[2,i]*res[p,inds[2,i]-1] +
          coef[0,i]*gauss[p,i];
    res[p,inds[0,i]-1] = (inds[1,i] == 0) ? tmp :
                coef[1,i]*res[p,inds[1,i]-1]+tmp ;
}  }
```

(a)

```
gauss^T = transpose(gauss); //real[d][u]
forall(p=0; p < u; p++) {
  res^T[inds[0,0]-1,p] = coef[0,0]*gauss^T[0,p];
  for(i=1; i<d; i++){
    tmp = coef[2,i]*res^T[inds[2,i]-1,p] +
          coef[0,i]*gauss^T[i,p];
    res^T[inds[0,i]-1,p] = (inds[1,i] == 0) ? tmp :
                coef[1,i]*res^T[inds[1,i]-1,p]+tmp ;
}  }
```

(b)

Fig. 7. "Imperative"-like code for Brownian bridge: original (a) and optimized for memory coalescing (b).

formula, optimizes memory usage by recording in array sobol only the current element of the scan. The previous iteration value is updated on line 6 in a reduction-like statement (i.e., sobol=sobol⊕b), but the rest of the code uses sobol outside reduction statements, which sequentializes the outer loop. The other pattern is demonstrated in the code on the right side by the loop for(..k..) res[k,j]=res[k-1,j]⊕b (lines 4 through 7), which has a cross-iteration dependency of distance 1. Both patterns correspond to scan(map) operators, which are difficult to recognize and further optimize via subscript (dependence) analysis. However, in our higher-level context, the compiler can produce more trivial outer-level parallelism by interchanging the scan and map by the following rule:

$$\texttt{scan}\quad(\texttt{map}\ \oplus) \equiv \texttt{transpose} \circ \texttt{map}\ (\texttt{scan}\ \oplus)\ \circ \texttt{transpose},$$

which says that "scanning" with a vectorized operator is semantically equivalent to mapping the scalar operator scan on the transposed matrix and transposing the result.

Furthermore, the original code presents other challenges to automatic parallelization, such as privatizable arrays that are indexed by induction variables that are conditionally incremented on only some of the loop's paths and form nonaffine subscripts that are difficult to analyze [Lin and Padua 2000; Oancea and Rauchwerger 2015].

However, there are code patterns and code transformations that are more suitably expressed in imperative rather than functional notation. The first case corresponds to dependent loops: Figure 7(a) shows a two-loop nest corresponding to the Brownian bridge implementation, in which the outer loop is parallel (semantically a map) but the inner loop is sequential—that is, each iteration computes a new element of res based on two other elements of res computed in (statically unknown) previous iterations and accessed via indirect array inds. Although not infrequent, such loops with constant-time array updates cannot be expressed purely functionally without resorting to sequential monadic code. The second case refers to lower-level optimizations. For example, the code in Figure 7(a) would result in noncoalesced access to global GPU memory because one GPU thread would compute an entire row of the result matrix—that is, the threads executing the same SIMD instruction would access memory references with a stride d, the size of the row. The solution, shown in Figure 7(b) is to compute the transposed result ($\texttt{res}^T$) and similarly to transpose the input array ($\texttt{gauss}^T$). The innermost index of these arrays is now p, which is the index of the parallel loop and results in coalesced access to global memory.

3.5. Optimizations Discussion

In essence, the reviewed optimizations seem to require a common ground between higher- and lower-level program representations. On the one hand, exploiting the higher-level semantics of parallel operators, such as streamPar, allows one to (1) design a fusion/fission engine that seems to scale at program level [Henriksen and Oancea 2013], (2) achieve coalesced accesses via transposition, and (3) encode in the program nontrivial (strength reduction) invariants. (Currently, all of these transformations are supported in Futhark, the language inspired by this benchmark.) In comparison, the validity of fusion/fission applied in a lower-level loop context requires sophisticated index-based (polyhedral) analysis, which does not always scale well with the size of the loop, and can be hindered by factors such as aliasing and flattened indices.

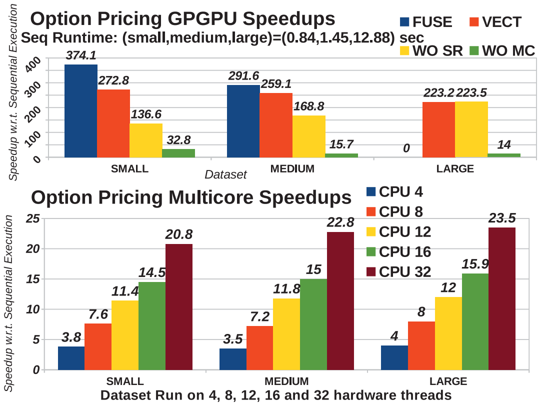
On the other hand, efficient execution of the sequential code often requires using loops that update in-place the array's elements of an array. This has motivated extending our language, Futhark, with support for in-place updates and normalized (do) loops. For example, in-place updates are supported via a mechanism relying on uniqueness types [Barendsen and Smetsers 1993; Henriksen 2014; Henriksen and Oancea 2013] and provide the guarantee that updating an array's element takes time proportional to the size of the element (rather than of the array). Together, in-place updates and loops allow (1) the user to efficiently write sequential code and (2) the compiler to efficiently sequentialize the excess parallelism and to support various important optimizations, such as hoisting and loop interchange and distribution.

3.6. Empirical Evaluation

The evaluation uses three datasets. The small dataset has parameters $\{n, u, d\} = \{8388608, 1, 1\}$. In other words, it uses 8,388,608 Monte Carlo paths to evaluate a vanilla-European call option, in which the payoff is the difference, if positive, between the value of underlying Dj Euro Stoxx 50 at a certain trigger date and a constant strike. A *discrete barrier option* is a contract with multiple trigger dates that forces the holder to exercise the option before maturity, whenever the underlyings cross specific barrier levels before one of the trigger dates. The medium dataset uses $n = 1{,}048{,}576$ paths to evaluate a discrete barrier option over $u = 3$ underlyings, namely the indexes Dj Euro Stoxx 50, Nikkei 225, and S&P 500, in which the payoff is a function of $d = 5$ trigger dates. The large dataset uses $n = 131{,}072$ iterations to evaluate a barrier option that is monitored daily ($d = 367$) and for which the payoff is conditioned on the barrier event and the market values, at exercise time, of the mentioned $u = 3$ underlyings.

The top and bottom bar graphs in Figure 8 show speedup results for the GPGPU and multicore-CPU execution, respectively. FUSE denotes the aggressively fused code version, and VECT denotes the version in which the map corresponding to the Monte Carlo iteration has been distributed. WOSR and WOMC denote the versions that do not use the strength-reduced Sobol formula and memory coalescing optimizations, respectively. CPU n denotes multicore execution on $n \in \{4, 8, 12, 16, 32\}$ hardware threads.

Table I provides information related to the dynamic behavior (run on multicore CPU) of the three program versions (fused, vectorized, and without strength reduction WOSR) on the three datasets (small (*sm*), medium (*md*), and large (*lg*): the total number of executed instructions (#Instr), the percentage of branches Br, memory reads Rd and writes Wr, and the miss ratio M for reads and writes (and total RW) for the first level of



Dataset	Small	Medium	Large
SeqTime (s)	0.84	1.45	12.88
$\mathrm{BW}^{\mathrm{GPU}}_{\mathrm{VECT}}$ (GB/s)	118.9	159.6	181.0
PercOptBW	41.2%	55.3%	62.8%
D↔H Mem (KB)	4.2	4.1	148.3
D↔H OV (μs)	226	241	357

`SeqTime`: the sequential CPU runtime.
$\mathtt{BW}^{\mathtt{GPU}}_{\mathtt{VECT}}$: the (global memory) bandwidth of the vectorized program (`VECT`), computed as the amount of global memory accessed divided by the kernel runtime.
`PercOptBW`: percentage achieved from optimal bandwidth (288.4 GB/s).
`D↔H Mem/OV` show the amount of memory and the runtime overhead corresponding to copying from device to host and back (including GPU allocation time).

Fig. 8. Option pricing speedups for GPGPU and multicore execution. FUSE and VECT denote the fused and distributed GPGPU code with all other optimizations on. WOSR and WOMC denote the absence of strength reduction/memory-coalescing optimizations. CPU n denotes multicore execution on n hardware threads. The uncertainty of the GPGPU runtime is between 0.1% and 0.7% of the runtime, with an average of 0.2%. The uncertainty of the multicore runtime is between 0.1% and 1.9% of the runtime, with an average of 0.7%.

Table I. Dynamic Characteristics of Three Program Versions Run on Multicore (CPU)

PRG_{set}	#Instr.	$\mathrm{Br}_{\%}$	$\mathrm{Rd}_{\%}$	$\mathrm{Wr}_{\%}$	$\mathrm{RW}_{\%}$	$\mathrm{M}^{L1}_{Rd\%}$	$\mathrm{M}^{L1}_{Wr\%}$	$\mathrm{M}^{L1}_{RW\%}$	$\mathrm{M}^{LL}_{Rd\%}$	$\mathrm{M}^{LL}_{Wr\%}$	$\mathrm{M}^{LL}_{RW\%}$	GMF
FUSE_{sm}	7.30E08	7.9	30.5	7.8	38.3	0.00	0.00	0.00	0.00	0.00	0.00	4KB
VECT_{sm}	8.80E08	11.4	36.6	17.5	54.2	1.11	0.97	1.06	0.22	0.50	0.31	64MB
WOSR_{sm}	1.68E09	14.2	34.3	12.2	46.5	0.00	0.01	0.00	0.00	0.00	0.00	4KB
FUSE_{md}	1.57E09	7.3	30.1	5.0	35.1	0.00	0.01	0.00	0.00	0.00	0.00	2KB
VECT_{md}	2.67E09	11.6	32.7	17.6	50.3	2.80	5.73	3.63	0.22	0.57	0.32	120MB
WOSR_{md}	2.72E09	12.2	32.1	16.8	48.9	0.00	0.01	0.00	0.00	0.00	0.00	2KB
VECT_{lg}	1.03E10	7.2	27.3	8.7	36.0	6.95	17.81	8.94	0.36	1.23	0.52	1.1GB
WOSR_{lg}	1.05E10	7.2	27.6	8.4	36.0	7.56	19.13	9.44	0.34	1.34	0.50	1.1GB

Note: The three code versions correspond to the fused (FUSE), vectorized (VECT), and without strength reduction optimization (WOSR) programs that are run on three datasets (i.e., small (*sm*), medium (*md*), and large (*lg*)). Columns 2 through 5 denote the total number of executed instructions and the percentage of branch (Br), memory read (Rd), and write (Wr) instructions, respectively. Columns 6 through 11 denote the cache miss ratio of reads (*Rd*), writes (*Wr*), and total (*RW*) for the first (*L*1) and last (*LL*) levels of cache, respectively. Finally, column 12 denotes the GPU global memory footprint.

cache `L1` and last level of cache `LL`, respectively. Finally, `GMF` is the GPGPU global memory footprint.

Comparing the fused and vectorized code versions, one can observe that the fused one executes on CPU (up to 1.7×) fewer instructions, exhibits a smaller percentage of memory reads and (especially) writes, and generates fewer cache misses at all levels.

In essence, the GPGPU fused version yields superior speedup because as long as the local arrays are small, they fit into the fast GPU memory, and global memory is not accessed on the critical path (small footprint `GMF`). As the size of the local arrays increases, each core consumes more of the sparse fast memory, resulting in a decreased GPU utilization. The medium dataset seems to capture the sweet spot: from there on, GPU VECT is winning, because its simpler kernels use fewer registers. Furthermore, the fused version cannot execute the large dataset, because there is not enough fast memory for each thread to hold 365×3 real numbers.

Comparing the fused with the version of the code that exclusively uses the independent Sobol formula (WOSR) on the small and medium datasets, one can observe that, as expected, strength reduction significantly decreases the dynamic-instruction count (up to 2.3× smaller), decreases the number of reads/writes to memory, and results in superior overall speedup. However, 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 on the large dataset, both the dynamic characteristics and the speedup achieved by the vectorized versions of the code, with and without strength reduction, are similar. Finally, memory coalescing is the most impactful optimization, being responsible for a speedup factor in the 10 to 20× range.

The table on the right-hand side of Figure 8 provides additional insight. Although all global memory accesses are coalesced, the (device-to-device) global memory bandwidth of the vectorized version $\mathtt{BW}^{\mathtt{GPU}}_{\mathtt{VECT}}$ is suboptimal because the program is not memory bound. For instance, the number of accesses to fast (shared and constant) memory is twice as high as the number of global memory accesses. The highest bandwidth (for the large dataset) corresponds to the lowest speedup; the fused version is not shown, as it does not use global memory on the critical path, resulting in highest speedup.

Finally, the amount of memory transferred between device and host (`D↔H Mem`) is small, in the hundreds of kilobytes range, because the whole computation is run on the GPU. However, the overhead is not negligible, ranging from 10% to 0.6% of the runtime of the small and large datasets, and is dominated by the overhead of the driver. In other words, the runtime is not proportional to the size of the transfer.

4. LOCAL VOLATILITY CALIBRATION

The presentation is organized as follows. Section 4.1 briefly states the financial problem and sketches its mathematical solution. Sections 4.2 and 4.3 present the code structure and the sequence of imperative transformations that are necessary to disambiguate and 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 beyond the knowledge of the common user and introduce (constant but) significant work overhead in comparison to the sequential code. Finally, Section 4.4 shows parallel CPU and GPU runtimes and demonstrates the tradeoff between efficient sequentialization and aggressive parallelization.

4.1. Financial and Mathematical Description

The pricing engine presented in Section 3 uses a Black-Scholes model and as such is limited to cases where the volatility can be assumed constant. More complex cases, such as when the contract has several payoff triggers, are more appropriately modeled by imposing their (local) volatility as a function of both time and current level of underlyings: $\sigma(t, S(t))$. In the following, we will focus on the case where the underlying is an equity stock without dividends, modeled under the risk-neutral measure by

$$dS(t) = r(t)S(t)dt + \sigma(t, S(t))S(t)dW(t), \tag{3}$$

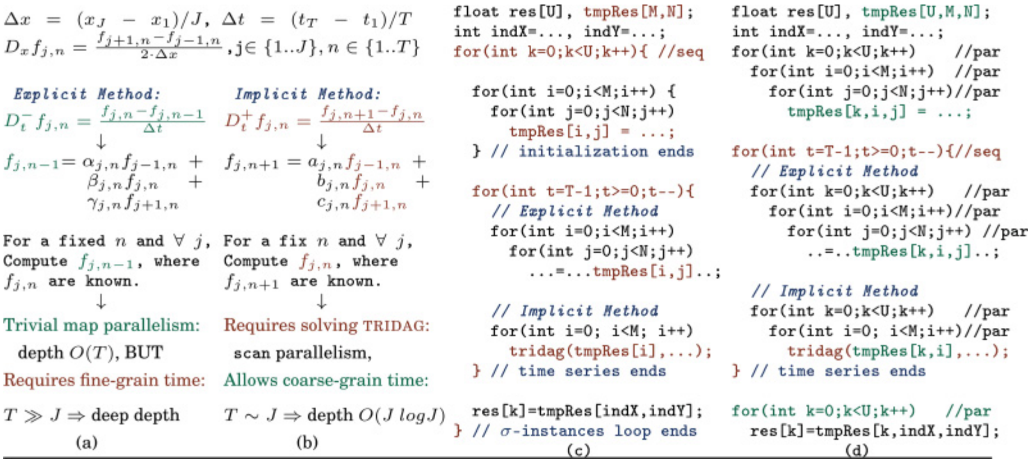
with instant volatility $\sigma(t, S(t))$ and where $r(t)$ is the risk-free rate and $W(t)$ is a Wiener process. The price function $f : \mathcal{S} \times [0, T] \to \mathbb{R}$, of a plain option on the preceding stock, is a solution of the PDE:

$$\frac{\partial f}{\partial t}(x,t) + r(t)x\frac{\partial f}{\partial x}(x,t) + \frac{1}{2}\sigma(x,t)^2x^2\frac{\partial^2 f}{\partial^2 x}(x,t) - r(t)f(x,t) = 0, \tag{4}$$

$$f(x, T) \;=\; F(x), \text{ where } (x,t) \in \mathcal{S} \times [0, T] \text{ and } F : \mathcal{S} \to \mathbb{R}, \tag{5}$$

with terminal condition expressed in terms of the known function F, which represents the payoff of the option at maturity. In the benchmark, the prices of a set of options with different strikes are hereby determined for a given specification of the σ function. Through this, one can heuristically calibrate the form and parameters of the σ function that best matches the current option prices observed in the market.[4] This section uses

[4] Standard practice is to compute local volatility more directly and faster by using the Dupire formula.



$\Delta x = (x_J - x_1)/J$, $\Delta t = (t_T - t_1)/T$
$D_x f_{j,n} = \frac{f_{j+1,n} - f_{j-1,n}}{2 \cdot \Delta x}$,j$\in \{1..J\}, n \in \{1..T\}$

Explicit Method:
$D_t^- f_{j,n} = \frac{f_{j,n} - f_{j,n-1}}{\Delta t}$
↓
$f_{j,n-1} = \alpha_{j,n} f_{j-1,n} + \beta_{j,n} f_{j,n} + \gamma_{j,n} f_{j+1,n}$

For a fixed n and $\forall\ j$, Compute $f_{j,n-1}$, where $f_{j,n}$ are known.
↓
Trivial map parallelism:
depth $O(T)$, BUT
Requires fine-grain time:
$T \gg J \Rightarrow$ deep depth
(a)

Implicit Method:
$D_t^+ f_{j,n} = \frac{f_{j,n+1} - f_{j,n}}{\Delta t}$
↓
$f_{j,n+1} = a_{j,n} f_{j-1,n} + b_{j,n} f_{j,n} + c_{j,n} f_{j+1,n}$

For a fix n and $\forall\ j$, Compute $f_{j,n}$, where $f_{j,n+1}$ are known.
↓
Requires solving TRIDAG:
scan parallelism,
Allows coarse-grain time:
$T \sim J \Rightarrow$ depth $O(J\ log J)$
(b)

```
float res[U], tmpRes[M,N];
int indX=..., indY=...;
for(int k=0;k<U;k++){ //seq

  for(int i=0;i<M;i++) {
    for(int j=0;j<N;j++)
      tmpRes[i,j] = ...;
  } // initialization ends

  for(int t=T-1;t>=0;t--){
    // Explicit Method
    for(int i=0;i<M;i++)
      for(int j=0;j<N;j++)
        ...=...tmpRes[i,j]..;

    // Implicit Method
    for(int i=0; i<M; i++)
      tridag(tmpRes[i],...);
  } // time series ends

  res[k]=tmpRes[indX,indY];
} // σ-instances loop ends
```
(c)

```
float res[U], tmpRes[U,M,N];
int indX=..., indY=...;
for(int k=0;k<U;k++)    //par
  for(int i=0;i<M;i++)  //par
    for(int j=0;j<N;j++)//par
      tmpRes[k,i,j] = ...;

for(int t=T-1;t>=0;t--){//seq
  // Explicit Method
  for(int k=0;k<U;k++)    //par
    for(int i=0;i<M;i++)//par
      for(int j=0;j<N;j++) //par
        ..=..tmpRes[k,i,j]..;

  // Implicit Method
  for(int k=0;k<U;k++)    //par
    for(int i=0; i<M;i++)//par
      tridag(tmpRes[k,i],...);
} // time series ends

for(int k=0;k<U;k++)    //par
  res[k]=tmpRes[k,indX,indY];
```
(d)

Fig. 9. Explicit (a) and implicit (b) methods and original (c) and transformed (d) code structure.

the material and notation from Munk [2007] to briefly recount the main steps of solving such an equation by a finite differences method [Crank and Nicolson 1947].

For simplicity, we discuss the case when $\mathcal{S} = \mathbb{R}$, but the benchmark uses a two-dimensional space discretization (i.e., $\mathcal{S} = \mathbb{R}^2$). The PDE system is solved by numerically approximating the solution with a sequence of difference equations, which are solved by sequentially iterating over the time discretization. The iteration starts from the known price F at time $t = T$ and moves backward toward $t = 0$. 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. The latter seemingly minor change results in very different algorithmic (work-depth) properties:

The explicit method, shown in Figure 9(a), approximates the time derivative backward by $D_t^- f_{j,n} = (f_{j,n} - f_{j,n-1})/\Delta t$, where $n \in 1..T$ and $j \in 1..J$ correspond to the discretized time and space. The resulting equation $f_{j,n-1} = \alpha_{j,n} f_{j-1,n} + \beta_{j,n} f_{j,n} + \gamma_{j,n} f_{j+1,n}$ directly computes the unknown values at time $n - 1$ from the values at time n. The latter are known since we move backward in time: from T toward 0. Although the space discretization can be efficiently, map-like, parallelized, the time series is inherently sequential by nature and results algorithmic depth because numerical stability requires the time to be finely grained discretized (i.e., T much larger than J).

The implicit method, shown in Figure 9(b), uses a forward-difference approximation for the time derivative, resulting in the equation $f_{j,n+1} = a_{j,n} f_{j-1,n} + b_{j,n} f_{j,n} + c_{j,n} f_{j+1,n}$, which says that the known value at time $n + 1$ equals a linear combination of unknown values at time n. The unknown values can be found by solving a tridiagonal system of equations (TRIDAG). The advantage of the implicit method is that it does not require particularly small timesteps, but the parallelization of the tridiagonal solver is beyond the knowledge of the common user, albeit it is possible via scans with linear-function composition and two-by-two matrix multiplication operators [Blelloch 1990], as explained in the next section. Since the scan parallelism has depth $O(\log J)$ for one time iteration, and the time and space discretization have comparable sizes, it follows that the total depth improves to $O(J \log J)$ when compared to the explicit method's $O(N)$, $N \gg J$. Finally, Crank-Nicolson combines the two approaches, does not require particularly small timesteps, converges faster, and is more accurate than the implicit method, albeit it still requires solving a tridiagonal system of equations.

4.2. Restructuring the Original Code for GPU Execution

Figure 9(c) shows the original structure of the code that implements volatility calibration in C pseudocode. The outermost loop of index `k=0..U-1` solves Equation (4) for a set of different strike prices. Here, the space is considered two dimensional, $S = \mathbb{R}^2$, and is traversed by loop indices `i=0..M-1` and `j=0..N-1`, on the y- and x-axes, respectively.

The body of the loop implements the Crank-Nicolson method and is formed by two loop nests. The first nest initializes array `tmpRes` in all points of the space discretization. The second nest corresponds to the time series that starts from the terminal condition in Equation (5) and moves toward 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 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 1 is reached, some of the points of interest of the space discretization are saved in array `res` (for each of the U loop iterations).

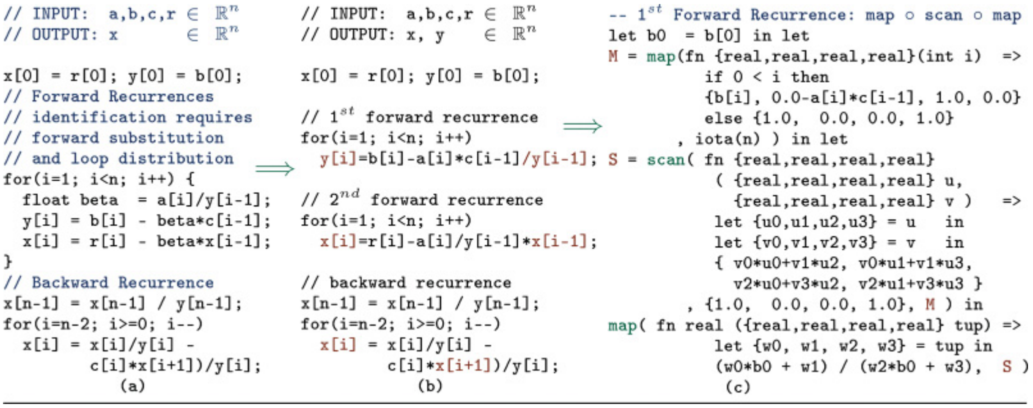
The remainder of this section describes the sequence of transformations that prepares the original code for efficient GPGPU execution; the result is shown in Figure 9(d).

The first difficulty corresponds to the outermost loop of Figure 9(c), which is annotated as sequential. The reason is that the space of the two-dimensional array `tmpRes[M,N]` 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 are 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 for GPGPU code generation; hence, array expansion comes to the rescue: the declaration of `tmpRes` is moved again outside the loop (to global memory), but it receives an extra dimension of size equal to U, the outermost loop count.

The second difficulty relates to the GPGPU programming model 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, after array privatization and expansion, the code offers significant nested parallelism; however, for example, the outermost loop of count `U` and the inner loop of count `M` cannot be both executed in parallel.[5] Perfect nests can be manufactured with two transformations: loop interchange and distribution. Although generally determining the legality of the transformations is nontrivial (e.g., using direction-vector based dependence analysis), matters are simple for parallel loops: a parallel loop (1) can be safely interchanged inward in a nest and (2) can be safely distributed across its statements. Figure 9(d) shows the resulting code after applying several loop distribution and interchange transformations:

[5] Although OpenCL and CUDA allow the user to express dynamic parallelism (device enqueue), the associated overhead is difficult to quantify and can be significant [Wang and Yalamanchili 2014]. For example, parent-child synchronization incurs significant runtime/memory footprint overhead, and the parent's shared memory is not visible in the child, and hence it needs to be transferred via global memory.



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

// 1^st forward recurrence
for(i=1; i<n; i++)
  y[i]=b[i]-a[i]*c[i-1]/y[i-1];

// 2^nd 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)
```

```
-- 1^st Forward Recurrence: map ○ scan ○ map
let b0  = b[0] in let
M = map(fn {real,real,real,real}(int i)  =>
          if 0 < i then
          {b[i], 0.0-a[i]*c[i-1], 1.0, 0.0}
          else {1.0,  0.0, 0.0, 1.0}
        , iota(n) ) in let
S = scan( fn {real,real,real,real}
           ( {real,real,real,real} u,
             {real,real,real,real} v )   =>
           let {u0,u1,u2,u3} = u   in
           let {v0,v1,v2,v3} = v   in
           { v0*u0+v1*u2, v0*u1+v1*u3,
             v2*u0+v3*u2, v2*u1+v3*u3 }
         , {1.0,  0.0, 0.0, 1.0}, M ) in
map( fn real ({real,real,real,real} tup) =>
           let {w0, w1, w2, w3} = tup in
           (w0*b0 + w1) / (w2*b0 + w3),  S )
            (c)
```

Fig. 10. TRIDAG: Original code (a), code after distribution (b), parallelization of first recurrence (c).

(1) The outermost loop of index `k` is distributed across the enclosed initialization and time-series loops. The initialization part now has the shape of a three-level perfect nest having degree of parallelism `U`×`M`×`N` and thus allowing efficient GPU utilization.
(2) However, the degree of parallelism has not yet been improved for the second nest, because the sequential time-series loop would separate the outermost and inner parallel loops. This is overcome by interchanging the outermost and the time-series loops.
(3) It is followed by distributing again the (former) outer loop of index `k` across the loop nests denoting the explicit and implicit methods. This results in two GPU kernels, of degree of parallelism `U`×`M`×`N` and `U`×`M`, which are executed inside the time-series loop.

The third difficulty corresponds to memory coalescing, which is solved whenever possible by interchanging loops or otherwise by transposing the "parallel" dimensions of the array inward, as discussed at the end of Section 3.4.

4.3. Parallelization of the Tridiagonal Solver (TRIDAG)

One can observe in Figure 9(d) that the implicit-method loop nest offers only two parallel loops that combine to a degree of parallelism equal to `U`×`M`, which might be too small on some datasets. In such cases, TRIDAG parallelization may improve matters.

Figure 10(a) shows the pseudocode for an implementation of TRIDAG. Both loops are sequential because the values of `x[i]` and `y[i]` in iteration `i` depend on the result of iteration `i-1`—that is, `x[i-1]` and `y[i-1]`. However, they can be automatically transformed to parallel code in four steps: (1) the local variable `beta` is forward substituted in both recurrences of the first loop; then (2) the first loop is distributed across its two remaining statements[6] and (3) the resulting one-statement recurrences, shown in Figure 10(b), 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, (4) the loops are replaced with parallel code.

For example, the recurrence $y_i = a_i + b_i / y_{i-1}$ can be brought to a parallel form by (1) first performing the change of variable $y_i \leftarrow q_{i+1}/q_i$, then (2) normalizing the obtained equation resulting in $q_{i+1} = a_i \cdot q_i + b_i \cdot q_{i-1}$, then (3) adding a trivial equation to form a system of two equations with two unknowns, which can be computed for all

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

Table II. Dynamic Characteristics of Two Program Versions Run on Multicore (CPU)

PRG_{set}	#Instr.	$\text{Br}_{\%}$	$\text{Rd}_{\%}$	$\text{Wr}_{\%}$	$\text{RW}_{\%}$	$\text{M}^{L1}_{Rd\%}$	$\text{M}^{L1}_{Wr\%}$	$\text{M}^{L1}_{RW\%}$	$\text{M}^{LL}_{Rd\%}$	$\text{M}^{LL}_{Wr\%}$	$\text{M}^{LL}_{RW\%}$	GMF
OUT_{sm}	1.17E10	5.7	38.0	7.7	45.7	1.27	5.71	2.04	0.00	0.01	0.00	3.5MB
ALL_{sm}	1.44E10	5.5	33.7	10.6	44.2	1.22	4.08	1.87	0.00	0.01	0.00	2.0MB
OUT_{md}	1.62E10	5.2	37.8	8.0	45.8	1.34	5.42	2.10	0.09	1.54	0.36	7.0MB
ALL_{md}	2.17E10	4.9	33.8	10.5	44.3	1.23	3.92	1.90	0.32	1.83	0.70	4.0MB
OUT_{lg}	2.37E11	5.7	38.4	8.6	47.0	1.55	8.70	2.91	0.19	2.26	0.59	112MB
ALL_{lg}	3.04E11	5.0	32.1	11.2	43.2	1.42	6.23	2.66	0.49	2.15	0.91	64MB

Note: OUT refers to the code version that uses the classical/sequential TRIDAG, and ALL refers to the one in which TRIDAG has been rewritten as a combination of three scans and six maps (but which are executed sequentially). These are run on three datasets (small (*sm*), medium (*md*) and large (*lg*)). Columns 2 through 5 denote the total number of executed instructions and the percentage of branch (Br), memory read (Rd), and write (Wr) instructions, respectively. Columns 6 through 11 denote the cache miss ratio of reads (*Rd*), writes (*Wr*), and total (*RW*) for the first (*L*1) and last (*LL*) levels of cache, respectively. Finally column 12 denotes the GPGPU global memory footprint.

$[q_{i+1}, q_i]$ vectors as a scan with a 2×2 matrix-multiplication (associative) operator:

$$\begin{bmatrix} q_{i+1} \\ q_i \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ 1.0 & 0.0 \end{bmatrix} * \begin{bmatrix} a_{i-1} & b_{i-1} \\ 1.0 & 0.0 \end{bmatrix} * \ldots * \begin{bmatrix} a_1 & b_1 \\ 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} b_0 \\ 1.0 \end{bmatrix}.$$

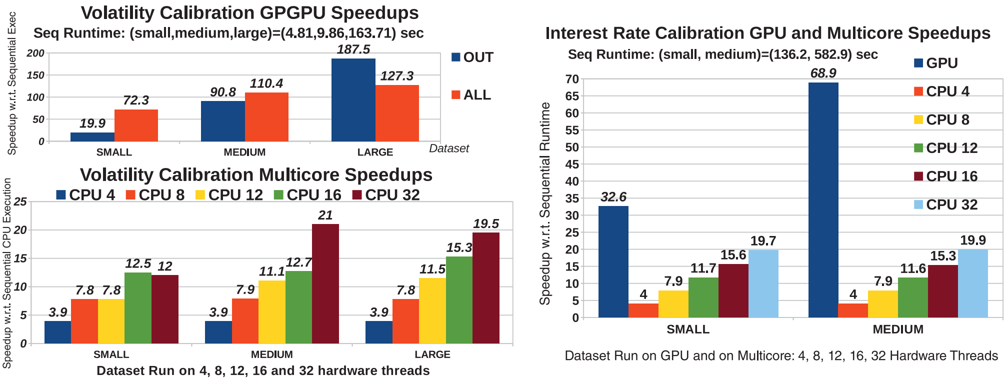
The corresponding parallel Futhark code is shown in Figure 10(c). Similarly, recurrence $x_i = a_i + b_i * x_{i-1}$ can be computed by a scan whose operator is a linear-function composition operator (also associative). However, exploiting TRIDAG's parallelism comes at a cost: it requires six map operations 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 if enough outer-level parallelism exists.

4.4. Empirical Evaluation

The evaluation uses three datasets. The small dataset has $\texttt{U}\times\texttt{M}\times\texttt{N}\times\texttt{T}= 16 \times 32 \times 256 \times 256$, and it favors the aggressive approach that parallelizes TRIDAG. The medium dataset has $\texttt{U}\times\texttt{M}\times\texttt{N}\times\texttt{T}= 128 \times 32 \times 256 \times 64$ and is intended to be a midpoint. The large dataset has $\texttt{U}\times\texttt{M}\times\texttt{N}\times\texttt{T}= 256 \times 256 \times 256 \times 64$ and contains enough parallelism in the two outer loops to utilize the hardware while enabling efficient sequentialization of TRIDAG.

Table II characterizes the dynamic behavior of the multicore code versions, denoted OUT and ALL, which correspond to the classical TRIDAG implementation and the one that has been rewritten to use (sequential) scans and maps. ALL executes up to 1.34× more instructions; this overhead is smaller than expected because ALL was amenable to aggressive vectorization. For example, the packed data-movement instructions MOVAPS, MOVHPS, and MOVLPS account for approximately 14.6% of ALL's instructions but only for approximately 1.2% of OUT's instructions. Additionally, ALL exhibits a higher percentage of write and a lower percentage of read operations than OUT and provides better locality for the first level of cache but slightly worse for the last level of cache. Finally, ALL's GPU global memory footprint (GMF) is 1.75× smaller than OUT because several intermediary arrays were stored in shared memory (this was possible because ALL exploits all parallelism).

The top and bottom bar graphs in Figure 11(a) show speedup results for the GPU and multicore execution, respectively. OUT and ALL denote the GPGPU code versions that execute TRIDAG sequentially and in parallel, respectively. As expected, OUT is significantly faster than ALL on the large dataset, where enough outer parallelism is available, whereas ALL performs much better on the small dataset, where the other method utilizes the hardware parallelism poorly. However, note that ALL uses an efficient intrablock segmented scan, which exploits the knowledge that the values of M and N are restricted to multiples of 32 less or equal to 256. The consequence is that a scanned segment never crosses the kernel's block boundaries, and ence scan executes



Volatility Calibration	Small	Medium	Large
Seq Runtime (s)	4.8	9.9	163.7
TR OV OUT/ALL (% Runtime)	6.9/25.9	11.6/14.3	16.4/11.0
BW$^{\mathtt{GPU}}$ OUT/ALL (% Optimal)	9.5/24.4	41.6/35.9	82.4/39.9
H⇒D / D⇒H Mem (bytes)	36/64	36/512	36/1024

(a)

Interest Rate	Small	Medium
Seq Runtime (s)	136.2	582.9
BW$^{\mathtt{GPU}}$ (% Optimal)	3.8	8.8
H⇒D / D⇒H Mem (KB)	3.3/13	3.3/9.3

(b)

Fig. 11. Parallel speedup of local volatility calibration (a) and interest rate calibration (b). OUT and ALL denote the GPGPU code versions that execute TRIDAG sequentially and in parallel, respectively. CPU n denotes multicore execution on n hardware threads. The uncertainty of GPU runtime is between 0.2% and 1.3% of the runtime, with an average of 0.7%. The uncertainty of multicore runtime is between 0.1% and 3.6% of the runtime, with an average of 0.6%. The tables present additional information: (1) Seq Runtime denotes the sequential CPU runtime in seconds; (2) TR OV OUT/ALL denotes the overhead of transposition as percentage from the parallel runtime for the OUT and ALL code versions, respectively; (3) BW$^{\mathtt{GPU}}$ denotes the global memory bandwidth of the GPU program as percentage from optimal hardware bandwidth (288.4%); and (4) H⇒D/D⇒H Mem denotes the amount of memory transferred from host to device and reverse, respectively (runtime overhead less than 1% of runtime).

entirely in fast memory and is nested in the kernel code. A general segmented scan implementation is less efficient. The multicore OpenMP version parallelizes only the outermost loop, which for the small dataset has only 16 iterations and results in only a 12× speedup on the 16 cores with two-way multithreading hardware.

Finally, the table in Figure 11(a) provides additional information related to the GPGPU execution. First, the transposition overhead, which is necessary to obtain coalesced access to global memory, ranges between around 7% and 25% of parallel runtime. Second, the bandwidth utilization for global memory reaches 82.4% of the peak hardware bandwidth for OUT on the large dataset. The other suboptimal values are either due to poor utilization of parallelism or due to the fact that the application is not memory bound (e.g., ALL versions aggressively utilize shared memory and use three scans). Third, the total memory transfer between host and device is less than 1KB (negligible overhead).

5. INTEREST RATE MODEL

The presentation is organized as follows. Section 5.1 presents the motivation for modeling the interest rate quickly and accurately. Section 5.2 presents the main components of a two-factor mean-reversion interest rate model and shows the top-level code structure. Finally, Section 5.3 discusses the difficulties raised by GPGPU parallelization and presents an evaluation of the parallel multicore and GPGPU running times.

5.1. 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, and cost of transactions, and is used to valuate at present time the future payoff of a given contract. Financial derivatives based on interest rates (e.g., swaps) are among the largest groups of derivatives exchanged on the global markets [Hull 2009]. The importance of interest rate models in financial computations has only increased with recent regulatory dispositions such as Basel III [Basel Committee on Banking Supervision 2010], requiring financial institutions to report the value of financial portfolios in different market scenarios. Before being employed, an interest rate model has to be calibrated. Its 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 accurate.

5.2. Financial Description

The interest rate calibration requires (1) an interest rate model, (2) a reference dataset with data observed in the market, (3) a bounded parameter space, (4) an error function measuring the divergence between the reference dataset and the output of a model based on a specific set of parameters, and (5) a search strategy for the identification of one or more optimal parameter sets. The short-rate model used here is the two-additive-factor Gaussian model (G2++) [Brigo and Mercurio 2006]. This model offers speed comparable to a single-factor model [Hull 2009], is more robust, and has been shown to better capture market volatilities.

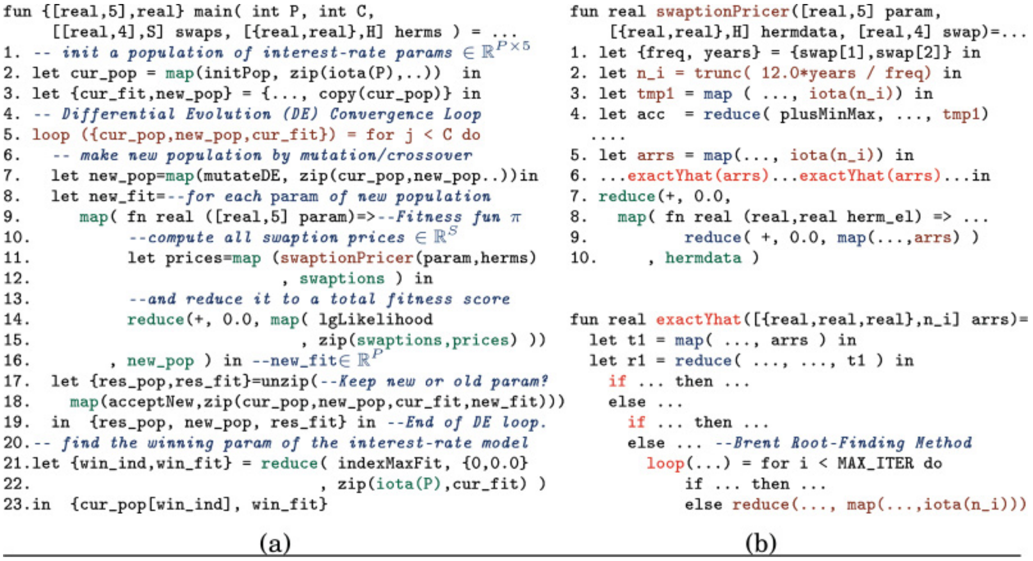
The G2++ model describes the short-term interest rate as a composition of two correlated stochastic processes and a deterministic time-dependent function. The model has five independent parameters $\texttt{param} = (\alpha, \beta, \sigma, \eta, \rho)$, which, once calibrated according to the present market consensus, can be employed in valuations. The two Gaussian processes are each described by a mean reversion constant (α, β) and a volatility term (σ, 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 $dx_t = -\alpha x_t dt + \sigma dW_t^1$ and $dy_t = -\beta y_t dt + \eta dW_t^2$ correlated by $\rho dt = dW_t^1 dW_t^2$. The third term, ϕ_t, is deterministic in time t, and the model will fit the discount factors observed in the market, if ϕ_t satisfies the following equation, where $f(0, T)$ denotes the instantaneous forward rate at time 0 for the maturity T currently observed in the market:

$$\phi_T = f(0, T) + \frac{\sigma^2}{2\alpha^2}(1 - e^{-\alpha T})^2 + \frac{\eta^2}{2\beta^2}(1 - e^{-\beta T})^2 + \rho\frac{\sigma\eta}{\alpha\beta}(1 - e^{-\alpha T})(1 - e^{-\beta T}).$$

With the param tuple fitted to the current market, a stochastic interest rate profile can be built for simulation of future scenarios. Additionally, an interest rate model calibrated to a given market can be used to price other instruments in the same market.

Our reference dataset, capturing the market consensus on the interest rate, consists of 196 European swaption[7] quotes, with constant swap frequency of 6 months and maturity dates and swap terms ranging from 1 to 30 years. The calibration process projects the dataset to one or more sets of param tuples. Since an inverse analytical relation is not available, the calibration is a search over a continuous five-dimensional parameter space. The parameter space is rugged so that minor updates in a param tuple would produce quite different interest rate scenarios. For the search to be effective,

[7] A *European interest rate swaption* is a contract granting its owner the nonbinding right to enter into an underlying swap with the issuer [Hull 2009]. This right is dependent on the level of the interest rate at the expiration date of the swaption. A *swap* is an agreement between two counterparties to exchange future cash flows, to reduce risk with the other party's comparative advantage in a different capital market.



```
fun {[real,5],real} main( int P, int C,
     [[real,4],S] swaps, [{real,real},H] herms ) = ...
1. -- init a population of interest-rate params ∈ ℝ^{P×5}
2. let cur_pop = map(initPop, zip(iota(P),..))  in
3. let {cur_fit,new_pop} = {..., copy(cur_pop)} in
4. -- Differential Evolution (DE) Convergence Loop
5. loop ({cur_pop,new_pop,cur_fit}) = for j < C do
6.    -- make new population by mutation/crossover
7.   let new_pop=map(mutateDE, zip(cur_pop,new_pop..))in
8.   let new_fit=--for each param of new population
9.      map( fn real ([real,5] param)=>--Fitness fun π
10.          --compute all swaption prices ∈ ℝ^S
11.          let prices=map (swaptionPricer(param,herms)
12.                         , swaptions ) in
13.          --and reduce it to a total fitness score
14.          reduce(+, 0.0, map( lgLikelihood
15.                            , zip(swaptions,prices) ))
16.       , new_pop ) in --new_fit∈ ℝ^P
17.   let {res_pop,res_fit}=unzip(--Keep new or old param?
18.     map(acceptNew,zip(cur_pop,new_pop,cur_fit,new_fit)))
19.   in  {res_pop, new_pop, res_fit} in --End of DE loop.
20.-- find the winning param of the interest-rate model
21.let {win_ind,win_fit} = reduce( indexMaxFit, {0,0.0}
22.                              , zip(iota(P),cur_fit) )
23.in  {cur_pop[win_ind], win_fit}
```

(a)

```
fun real swaptionPricer([real,5] param,
    [{real,real},H] hermdata, [real,4] swap)=...
1. let {freq, years} = {swap[1],swap[2]} in
2. let n_i = trunc( 12.0*years / freq) in
3. let tmp1 = map ( ..., iota(n_i)) in
4. let acc  = reduce( plusMinMax, ..., tmp1)
  ....
5. let arrs = map(..., iota(n_i)) in
6. ...exactYhat(arrs)...exactYhat(arrs)...in
7. reduce(+, 0.0,
8.   map( fn real (real,real herm_el) => ...
9.          reduce( +, 0.0, map(...,arrs) )
10.      , hermdata )


fun real exactYhat([{real,real,real},n_i] arrs)=
  let t1 = map( ..., arrs ) in
  let r1 = reduce( ..., ..., t1 ) in
    if ... then ...
    else ...
      if ... then ...
      else ... --Brent Root-Finding Method
        loop(...) = for i < MAX_ITER do
            if ... then ...
            else reduce(..., map(...,iota(n_i)))
```

(b)

Fig. 12. Code Structure of the top-level interest rate calibration (a) and its pricing component (b).

an efficient exploration of the parameter space is paramount, as well as a method to quickly price market contracts according to the interest rate behavior dictated by param. Figure 12(a) shows the top-level structure of the implementation.

As a search heuristic, the algorithm uses the Markov chain Monte Carlo differential evolution (DE-MCMC)[8] method [Storn and Price 1997; Braak 2006]. As in a genetic algorithm, a population of P param tuples is randomly initialized within the continuous parameter space at line 2, and the convergence loop at line 5 "evolves" the population toward one that better fits the observed market prices. Each iteration builds a new population at line 7 by either mutating or replacing part of the old population with a recombination of surviving candidates. The speed of the search can be tuned with the ratio between mutations and replacements, as well as with the amplitude of the mutations.

The quality of each individual is measured by a fitness function π, implemented by the unnamed-function parameter of the map at lines 9 through 16. For each candidate param, all swaptions are priced according to the G2++ model (Chapter 4 in Brigo and Mercurio [2006]) (lines 11 and 12), and then the map-reduce composition at lines 14 and 15 computes a total fitness score, which summarizes the differences between the observed marked prices and the param-modeled prices. We do not describe the swaption pricing and the fitness function in more detail here, but we mention that the heart of the pricer involves the use of Brent's root-finding method (Chapter 4 in Brent [1973]) and the computation of an integral using the Gauss-Hermite quadrature technique [Steen et al. 1969]. Finally, Markov chain selection is implemented at lines 17 and 18, where the params of the new population are accepted or rejected by comparing their old and new fitness scores. After convergence is reached, the fittest param is found and returned at lines 21 through 23.

[8] DE-MCMC is a form of Bayesian analysis that allows to estimate both an optimal solution (returned by the differential equation) and the amount of clusters of optimal solutions. The latter is important in a statistical context.

Table III. Dynamic Characteristics Run on Multicore (CPU)

Dataset	#Instr.	$Br_{\%}$	$Rd_{\%}$	$Wr_{\%}$	$RW_{\%}$	$M^{L1}_{Rd\%}$	$M^{L1}_{Wr\%}$	$M^{L1}_{RW\%}$	$M^{LL}_{Rd\%}$	$M^{LL}_{Wr\%}$	$M^{LL}_{RW\%}$	GMF
Small	4.42E11	8.2	43.6	18.1	61.8	0.03	0.05	0.03	0.00	0.03	0.01	6.9MB
Medium	1.59E12	8.2	45.2	17.3	62.5	0.02	0.06	0.03	0.00	0.04	0.01	9.5MB

Note: Columns 2 through 5 denote the total number of executed instructions and the percentage of branch (Br), memory read (Rd), and write (Wr) instructions, respectively. Columns 6 through 11 denote the cache miss ratio of reads (*Rd*), writes (*Wr*), and total (*RW*) for the first (*L*1) and last (*LL*) levels of cache, respectively. Finally, column 12 denotes the GPGPU global memory footprint.

5.3. Empirical Evaluation

The implementation depicted in Figure 12(a) reveals an outer level of balanced map parallelism, such as the one that independently applies the fitness function π to each of the P param tuples of the population at line 9. For front-office purposes, P typically varies between 64 and 256, which is not enough to fully utilize the GPU, whereas a higher P could be useful in risk control to cluster params in offline multiscenario analyses. The next level of parallelism, corresponding to pricing each of the S=196 swaptions at line 11, presents very unbalanced work. The swaption pricer, shown in Figure 12(b), starts by computing a time discretization at line 2, whose size n_i is sensitive to the current swaption and ranges between 2 and 60. Then it maps and reduces many arrays of (irregular) size n_i within very nested/rich control flow (e.g., see the function exactYhat). Sequentializing the pricer is not an option because of the prohibitive cost of divergent execution on GPU. Flattening all parallelism [Blelloch et al. 1994] would still be inefficient because (1) flattening the control flow would introduce many scatter-gather operations, which are very expensive to compute on GPU (and tedious to write by hand), and (2) because of the overhead of segmented versions of scan and reduce.

Our current implementation is a midpoint between the two approaches in that it packs as many irregular arrays inside a GPU block as possible without crossing the block boundaries and “turns-off” the remaining threads. This has the advantage that it does not introduces scatter-gather operations and that the segmented reduces operate entirely in local memory and does not exit to CPU, but the disadvantage of introducing redundant computation and register pressure. The results for the parallel GPGPU and multicore speedup were shown in Figure 11(b). Although in this case the GPU execution is couple of times faster than the parallel CPU execution, the speedup is significantly lower than the other two applications, and further optimization remains an open challenge. Finally, Table III provides information related to the characteristics of the multicore execution, which features about 8% and 62% of branch and memory instructions, as well as good locality at both the first and last level of cache.

6. RELATED WORK

There are several strands of related work. First, a considerable amount of work has targeted (1) the parallelization of financial computations on many-core hardware [Joshi 2010; Lee et al. 2010; Oancea et al. 2012]; (2) production integration in large banks’ IT infrastructure, such as for efficient, end-to-end pricing of exotic options [Nord and Laure 2011]; or (3) integrating a contract-specification language [Bahr et al. 2015] and dynamical graphical user interfaces [Elsman and Schack-Nielsen 2014] within a language-heterogeneous financial framework using type-oriented techniques. Such difficulties have been experienced in other compute-intensive areas as well, such as in interoperating across computer-algebra systems, and have led to point-to-point but ultimately nonscalable solutions [Chicha et al. 2004; Oancea and Watt 2005].

Our work differs in that we seek to express parallel patterns as high-level functional constructs with the aim of systematically (and automatically) generating efficient parallel code. Futhark [Henriksen et al. 2014; Henriksen and Oancea 2013; Henriksen

2014] is an ongoing effort at providing an intermediate-language tool chain to which production-level (functional) languages (e.g., APL) could be compiled.

Second, a large body of related work is concerned with autoparallelization of imperative code. Such work includes static dependency analyses [Feautrier 1991; Pouchet et al. 2011; Lin and Padua 2000] for recognizing and extracting loop-level parallelism, as well as for characterizing/improving memory access patterns [Elango et al. 2015]. Other techniques may extract partial parallelism dynamically [Dang et al. 2002; Oancea et al. 2005], but such techniques have not been evaluated (yet) on GPGPUS.

A third strand of related work covers techniques for achieving GPGPU high performance [Ryoo et al. 2008], which involves achieving memory coalescing via block tiling, optimizing register usage via loop unrolling, and 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 [Bell and Hoberock 2011]. Implementation of these principles as compiler optimizations ranges from (1) heuristics based on pattern matching [Dubach et al. 2012; Ueng et al. 2008; Yang et al. 2010], over (2) more formal modeling of affine transformations via the polyhedral model [Amini et al. 2011; Baskaran et al. 2010], to (3) aggressive techniques, such as loop collapsing, that may be applicable even for irregular control flow and memory access patterns [Lee et al. 2009].

A large body of related work includes DSLs for programming many-core systems, which are either embedded in Haskell, such as Nikola [Mainland and Morrisett 2010], Accelerate [Chakravarty et al. 2011], Obsidian [Claessen et al. 2012], and SPL [Flænø Werk et al. 2012], or stand-alone, as in SAC [Grelck and Scholz 2006; Guo et al. 2011]. Such solutions do not typically support nested parallelism.

Also related to the present work is the work on array languages in general (including APL [Iverson 1962] and its derivatives) and the work on capturing the essential mathematical algebraic aspects of array programming [Hains and Mullin 1993] and list programming [Bird 1987] for functional parallelization. Compilers for array languages also depend on inferring shape information either dynamically or statically [Elsman and Dybdal 2014], although they can often assume that the arrays operated on are regular, which is not the case for Futhark programs.

A scalable technique for targeting parallel architectures in the presence of nested parallelism is to apply Blelloch's flattening transformation [Blelloch 1996]. Blelloch's technique has also been applied in the context of compiling NESL [Bergstrom and Reppy 2012] 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 [Madsen and Filinski 2013], which supports a stream data type 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, particularly 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 [Standard Performance Evaluation Corporation 2014] provided by the Standard Performance Evaluation Committee (SPEC). Compared to the present work, the SPEC ACCEL benchmark contains few, if any, applications related to the financial domain, and further, the goal of the SPEC ACCEL benchmark is not to demonstrate that different utilization of parallelism can be appropriate for different input datasets. Finally, Joshi et al. [2006] present a methodology for clustering programs based on the similarity of their inherent characteristics, such as dynamic instruction mix, register dependency distance (as a measure of ILP), and spatial and temporal locality. We note that each of our three benchmarks is likely to be in a different equivalence class of programs, as they exhibit a very different dynamic mix of instructions and LL_1 miss ratio. This extends even to variants of the same benchmark, as in the case of option pricing's `FUSE` and `VECT` versions.

7. CONCLUSION AND FUTURE WORK

This article has presented three real-life financial applications that constitute a challenging and compelling application for optimizing compilers because they exhibit a rich, data-sensitive optimization space whose manual exploration is simply too tedious. We have demonstrated how parallelism and related invariants can be expressed in a generic functional notation, and how various trade-offs can be exploited by high-level code transformations, which ultimately result in low-level code versions that exhibit very different runtime behaviors, albeit each of them is better suited on a certain class of datasets. Finally, empirical measurements have demonstrated the feasibility of the proposed transformations, the trade-offs, and the difficulties.

REFERENCES

Mehdi Amini, Fabien Coelho, Francois Irigoin, and Ronan Keryell. 2011. Static compilation analysis for host-accelerator communication optimization. In *Proceedings of the Conference on Languages and Compilers for Parallel Computing (LCPC'11)*. 237–251.

Patrick Bahr, Jost Berthold, and Martin Elsman. 2015. Certified symbolic management of financial multi-party contracts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*.

Erik Barendsen and Sjaak Smetsers. 1993. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, Vol. 761. Springer, 41–51.

Basel Committee on Banking Supervision. 2010. *Basel III: A Global Regulatory Framework for More Resilient Banks and Banking Systems*. Bank for International Settlements, Basel, Switzerland.

M. M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction (CC'10)*. 244–263.

Nathan Bell and Jared Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition*, W.-M. W. Hwu (Ed.). Morgan Kaufmann, San Francisco, CA.

Lars Bergstrom and John Reppy. 2012. Nested data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*. 247–258.

R. S. Bird. 1987. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study on Logic of Programming and Calculi of Discrete Design*. 5–42.

F. Black and M. Scholes. 1973. The pricing of options and corporate liabilities. *Journal of Political Economy* 81, 3, 637–654.

Guy Blelloch. 1996. Programming parallel algorithms. *Communications of the ACM* 39, 3, 85–97.

Guy E. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers* 38, 11, 1526–1538.

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Carnegie Mellon University, Pittsburgh, PA.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing* 21, 1, 4–14.

Cajo J. Braak. 2006. A Markov chain Monte Carlo version of the genetic algorithm differential evolution: Easy Bayesian computing for real parameter spaces. *Statistics and Computing* 16, 3, 239–249.

Paul Bratley and Bennett L. Fox. 1988. Algorithm 659 implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software* 14, 1, 88–100.

Richard P. Brent. 1973. *Algorithms for Minimization without Derivatives*. Prentice Hall.

Damiano Brigo and Fabio Mercurio. 2006. *Interest Rate Models—Theory and Practice: With Smile, Inflation and Credit* (2nd ed.). Springer.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the 6th Workshop on Aspects of Multicore Programming (DAMP'11)*. 3–14.

Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. 2004. Parametric polymorphism for computer algebra software components. In *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.* 119–130.

Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming (DAMP'12)*. 21–30.

J. Crank and P. Nicolson. 1947. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society* 43, 1, 50–67.

Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the International Parallel and Distributed Processing Symposium (PDPS'02)*. 20–29.

Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a high-level language for GPUs. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'12)*. 1–12.

Daniel Egloff. 2011. Pricing financial derivatives with high performance finite difference solvers on GPUs. In *GPU Computing Gems Jade Edition,* W.-M. W. Hwu (Ed.). Morgan Kaufmann, San Francisco, CA, 309–322.

V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. 2015. On characterizing the data access complexity of programs. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, NY, 567–580.

Martin Elsman and Martin Dybdal. 2014. Compiling a subset of APL into a typed intermediate language. In *Proceedings of the 1st International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY.

Martin Elsman and Anders Schack-Nielsen. 2014. Typelets—a rule-based evaluation model for dynamic, statically typed user interfaces. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL'14)*.

Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1, 23–54.

Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. 2012. An embedded DSL for stochastic processes: Research article. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'12)*. ACM, New York, NY, 93–102.

M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. 2011. Performance analysis and optimisation of the OP2 framework on many-core architectures. *ACM SIGMETRICS Performance Evaluation Review* 38, 4, 9–15.

Paul Glasserman. 2004. *Monte Carlo Methods in Financial Engineering*. Springer, New York, NY.

Clemens Grelck and Sven-Bodo Scholz. 2006. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming* 34, 4, 383–427.

Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11).* ACM, New York, NY, 15–24.

G. Hains and L. M. R. Mullin. 1993. Parallel functional programming with arrays. *Computer Journal* 36, 3, 238–245.

Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 2005. Interprocedural parallelization analysis in SUIF. *ACM Transactions on Programming Languages and Systems.* 27, 4, 662–731.

Troels Henriksen. 2014. *Exploiting Functional Invariants to Optimise Parallelism: A Dataflow Approach*. Master's Thesis. DIKU, Copenhagen, Denmark.

Troels Henriksen, Martin Elsman, and Cosmin Eugen Oancea. 2014. Size slicing—a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'14)*. ACM, New York, NY, 31–42.

Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'13)*. ACM, New York, NY, 47–58.

Troels Henriksen and Cosmin Eugen Oancea. 2014. Bounds checking: An instance of hybrid analysis. In *Proceedings of the ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, 88.

Roger W. Hockney. 1965. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM* 12, 1, 95–113.

J. Hull. 2009. *Options, Futures and Other Derivatives*. Prentice Hall.

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons.

Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy Kurian John. 2006. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactios on Computers* 6, 769–782.

M. S. Joshi. 2010. Graphical Asian options. *Wilmott Journal* 2, 2, 97–107.

Hee-Seok Kim, Shengzhao Wu, Li-Wen Chang, and Wen-Mei W. Hwu. 2011. A scalable tridiagonal solver for GPUs. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE, Los Alamitos, CA, 444–453.

A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes. 2010. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *Journal of Computational and Graphical Statistics* 19, 4, 769–789.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. 101–110.

Yuan Lin and David Padua. 2000. Analysis of irregular single-indexed arrays and its applications in compiler optimizations. In *Proceedings of the International Conference on Compiler Construction*. 202–218.

Frederik M. Madsen and Andrzej Filinski. 2013. Towards a streaming model for nested data parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*.

Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the 3rd ACM International Symposium on Haskell*. 67–78.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

Claus Munk. 2007. Introduction to the Numerical Solution of Partial Differential Equations in Finance. Retrieved May 10, 2016, from http://mit.econ.au.dk/vip_htm/cmunk/noter/pdenote.pdf.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

Fredrik Nord and Erwin Laure. 2011. Monte Carlo option pricing with graphics processing units. In *Proceedings of the International Conference on Parallel Computing (ParCo'11)*.

Cosmin Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial software on GPUs: Between Haskell and Fortran. In *Proceedings of the Workshop on Functional High-Performance Computing (FHPC'12)*. ACM, New York, NY, 61–72.

Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable conditional induction variable (CIV) analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'15)*.

Cosmin E. Oancea, Jason W. A. Selby, Mark Giesbrecht, and Stephen M. Watt. 2005. Distributed models of thread level speculation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, Vol. 5. 920–927.

C. E. Oancea and S. M. Watt. 2005. Domains and expressions: An interface between two approaches to computer algebra. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation (ISSAC'05)*. ACM, New York, NY, 261–269.

L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 549–562.

James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media.

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-Mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. 73–82.

Standard Performance Evaluation Corporation. 2014. SPEC ACCEL. Retrieved May 10, 2016, from https://www.spec.org/accel/.

N. M. Steen, G. D. Byrne, and E. M. Gelbard. 1969. Gaussian quadratures for the integrals $\int_0^\infty \exp(-x^2) f(x)dx$ and $\int_0^b \exp(-x^2) f(x)dx$. *Mathematics of Computation* 23, 661–671.

Rainer Storn and Kenneth Price. 1997. Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization* 11, 4, 341–359.

Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. 2008. CUDA-lite: Reducing GPU programming complexity. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*. 1–15.

Jin Wang and Sudhakar Yalamanchili. 2014. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'14)*. 51–60.

David Watkins. 1991. *Fundamentals of Matrix Computations*. Wiley, New York, NY.

M. J. Wichura. 1988. Algorithm AS 241: The percentage points of the normal distribution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 37, 3, 477–484.

Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*. 86–97.

Received August 2015; revised February 2016; accepted February 2016