Financial Software on GPUs: Between Haskell and Fortran

Cosmin E. Oancea[1], Christian Andreetta[1], Jost Berthold[1], Alain Frisch[2], Fritz Henglein[1]

HIPERFIT, [1]Department of Computer Science, University of Copenhagen (DIKU) and [2]LexiFi

cosmin.oancea@diku.dk, christian.andreetta@diku.dk, berthold@diku.dk, alain.frisch@lexifi.com, henglein@diku.dk

Abstract

We present a case study, in which we optimize the execution time of a relatively complex single-core high-performance commercial pricing kernel for financial derivatives by parallelizing it and systematically hand-compiling it to OpenCL, resulting in speedups as high as $\sim 70\times$ on a commodity laptop graphical processor unit (GPU) and $\sim 500\times$ on a mid-range desktop GPU. The parallelization consists of specifying the kernel in a completely hardware-independent functional form that employs map-reduce-scan combinators for expressing data parallelism, and step-by-step hand-translating it to OpenCL code by employing generally and independently applicable compiler transformations.

Apart from the concrete speed-ups attained, our contributions are twofold: First, from a *language perspective*, we illustrate that even state-of-the-art auto-parallelization techniques are incapable of discovering all the requisite data parallelism when rendering the functional code in Fortran-style imperative array processing form. Second, from a *performance perspective*, we study which compiler transformations are necessary to map the high-level functional code to hand-optimized OpenCL code for GPU execution. We discover a rich optimization space with nontrivial trade-offs and cost models. Memory reuse in map-reduce patterns, strength reduction, branch divergence optimization, and memory access coalescing each exhibit significant impact by themselves. Combined they facilitate essentially full utilization of all GPU cores.

Functional programming has played a crucial double role in our case study: Capturing the naturally data-parallel structure of the pricing algorithm in a transparent, reusable and entirely hardware-independent fashion; and supporting the correctness of the subsequent compiler transformations to a hardware-oriented target language by a rich class of universally valid equational properties. Taking the observed difficulty of automatic parallelization of imperative sequential code and inherent labor of porting hardware-oriented programs into account, our case study suggests the possibility of functional programming technology facilitating *high-level* expression of leading-edge performant *portable* high-performance systems for massively parallel hardware architectures.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

General Terms Performance, Design, Algorithms

Keywords autoparallelization, tiling, memory coalescing, strength reduction, functional language

1. Introduction

The financial system is facing fundamental challenges because of their complexity, interconnectedness and speed of interaction. International banking and insurance regulations increasingly focus on analyzing and reducing the systemic effects of financial institutions on the financial system as a whole. For this reason, such institutions are asked to evaluate their reliability and stability in a large number of economic scenarios, with some of the scenarios presenting critical conditions that require large scale modeling efforts. In this context, Monte Carlo simulations, originally developed by physicists to efficiently investigate the stochastic behavior of complex, multidimensional spaces, have emerged as tools of choice in critical applications like risk modeling and pricing of financial contracts. These simulations are paradigmatic *Big Compute* problems that transcend the domain of *embarrassingly parallel* problems. From a hardware architecture perspective, they require employing and effectively exploiting massive *parallelism*. Interesting results have been achieved by efficient management of processes on grid farms and expert use of specialized hardware such as graphic processing units (GPUs) [47]. In particular, the latter unite the advantages of parallelization, low power consumption, and low latency in data transfer to efficiently execute a large number of single instructions on multiple data (SIMD). This kind of massively parallel hardware requires programming practices that differ from conventional imperative von-Neumann-machine-style programming, however.

The desirability of a programming model that supports high-level description of large-scale data transformations for modeling purposes, coupled with the need to target rapidly evolving massively parallel hardware architectures without letting these infiltrate the programs themselves has led us to concentrate on the well-established practices of *functional programming*. Functional languages are renowned for their good modularity, testability and code reuse [25], which drastically improves maintainability and transparency – crucial properties in areas where the success of a company depends on the correctness and reliability of its software. Furthermore, the purity of functional languages largely facilitates reasoning about the inherent parallelism of an algorithm, and effective parallelizations exist for common higher-order functions [23].

Functional languages are increasingly employed in financial institutions for modeling and high-productivity programming purposes, for instance DSLs for finance [3, 38]. At the same time, research on functional programming models targeting novel hardware (such as GPUs and FPGAs) has demonstrated their ability to exploit the hardware without letting its specifics encroach on the programming model [12, 30]. It is the double match of functional programming with modeling in quantitative finance and with naturally expressing data parallelism that motivates our research into architecture-independent parallelization of financial code using a functional approach.

In the remainder of this section we provide a rationale for our case study and an overview of the optimization techniques evaluated. In the following sections we present the functional formulation of the pricing algorithm (Section 2), the optimizations for compiling it to OpenCL (Section 3), the experimental results of implementing the optimizations on a low-end commodity laptop graphics card (Section 4), a review of related work on imperative and functional parallelization (Section 5), and finally our conclusions as to what has been accomplished so far and which future work this suggests (Section 6).

1.1 Bird's Eye View

While speeding up the runtime of financial software by hand-parallelizing the code for GPU execution is in itself of pragmatic importance, this paper takes a broader view, in which we use the gained insights to evaluate the language and compiler infrastructure needed to automate the process. The main objectives are twofold:

Language. We take the perspective that the language should provide what is necessary for the user (i) to express algorithmic invariants explicitly in the language, and, in general, (ii) to write an implementation that comes as close as possible to the "pure" algorithmic form. If the algorithm is inherently parallel, then we expect the implementation to preserve this property. In this sense, without having parallelism in mind, we have written a sequential, functional (Haskell) version of the generic-pricing algorithm to provide a baseline for comparison against the original imperative (`C`) code.

Not surprisingly, we find that the functional style, with better support for mathematical abstraction, makes parallelism (almost) explicit by means of higher-order functions such as `map`, `fold` and `scan` (i.e., `do-all`, reduction and prefix sum). On the other hand, "good imperative-programming practice" seems to inherently result in code that is optimized for sequential execution, but obfuscates the inherent algorithmic parallelism to an extent that makes it difficult to recognize for both programmer and compiler.

We demonstrate this perspective throughout the paper by presenting side-by-side examples of imperative *vs.* functional code and surveying the vast literature of autoparallelizing techniques. Section 1.2 highlights the programming-style differences via a contrived, but still illustrative, example.

Performance. While we have argued that algorithmic clarity should come first, we also take the view that this should not be achieved by compromising performance. The second objective of this paper, outlined in Section 1.3, is to explore the compiler optimizations that have proved most effective for our case study, although they have been implemented by hand:

- We present evidence of how user-specified invariants can drive powerful high-level optimizations (e.g. strength reduction).
- We reveal a rich optimization space that exhibits non-trivial cost models, which are best left in the care of the compiler.
- We discuss several lower-level, GPU-related optimizations that have to be the compiler's responsibility if we require hardware transparency (i.e. write once - run anywhere).

1.2 Language Perspective

Figure 1 presents two semantically-equivalent functions, written in Fortran77 and Haskell, which are our instances of imperative and functional languages, respectively. With Haskell, we disregard lazyness issues and use simple lists instead of performance-oriented special types like vectors or arrays for clarity. The example is telling in that it combines several interesting coding patterns that

```
CC FORTRAN CODE
1  SUBROUTINE example ( D, N, M, dirVs, ret )
2    INTEGER i, j, k, D, N, M, len
3    INTEGER ia(M), ret(D,N), dirVs(M,D)
4    DO i = 1, N
5      len = 0
6      DO k = 1, M
7        IF( test(i,k) ) THEN
8          len     = len + 1
9          ia(len) = k
10     ENDIF  ENDDO
11     DO j = 1, D
12       ret(j, i) = 0
13       DO k = 1, len
14         ret(j,i) = ret(j,i) XOR dirVs(ia(k), j)
15       ENDDO
16       IF(i .GT. 1)
17         ret(j,i) = ret(j,i) XOR ret(j,i-1)
18     ENDDO
19 ENDDO END

-- HASKELL CODE
20 example :: Int -> Int -> Int -> [[Int]] -> [[Int]]
21 example n m dirVs =          --  d × m     n × d
22   let lbody:: Int -> [Int]
23       lbody i =
24         let ia     = filter (test i) [0..m-1]
25             xorV v = fold xor 0 [v!j | j<-ia]
26         in map xorV dirVs
27       ret = map lbody [1..n]
28       e   = replicate (length dirVs) 0
29   in tail (scan (zipWith xor) e ret)
```

Figure 1. Contrived, but illustrative example: Fortran77 *vs* Haskell

appear in implementations of Sobol quasi-random sequences [10], and contrived in that it does not produce random numbers.

Haskell Code. Let us examine first the `lbody` function at lines $22-26$: Indexes in $0..m-1$ are filtered based on the `test` predicate, e.g., testing whether index $k \in [0..m-1]$ in the bit-representation of i is set. Next, (i) the `xorV` function reduces the elements corresponding to the filtered indexes of a `dirVs`'s row with the `xor` operator (i.e., `fold` at line 25), and (ii) this is applied to each row of `dirVs`, i.e., the `map` at line 26. The result of `lbody` is thus a list of the same length (denoted `d`) as `dirVs`.

The rest of `example`'s implementation is straightforward: (i) at line 27 `lbody` is mapped to each integer in $[1..n]$, resulting in a list representation of a $n \times d$ matrix, named `ret`, and finally (ii) prefix-sum with operator `xor` is applied to aggregate the elements in the same position in each row of `ret`, i.e., the `scan` at line 29.

One can observe that parallelism is made (almost) explicit in the implementation by the sequence of `map` and `scan` at lines 27 and 29. The latter has depth $log(n)$, while the former is embarrassingly parallel and exhibits nested[1] parallelism that could be further optimized via flattening [7, 11].

Fortran Code. Examining the Fortran code, an experienced imperative programmer might recognize that (i) the `do k` loop at lines $6-10$ implements the filtering of indexes based on the `test` predicate, and (ii) the `do k` loop at lines $13-15$ corresponds to the `fold` at line 25. (Note that Fortran uses column-major arrays). The outermost loop and the `do j` loop at lines $11-18$ (minus line 17) correspond to the Haskell `map`s at lines 27 and 26, which compute the result array `ret`. The code is arguably less obvious than the one in Haskell, due to the lack of higher-order functions such as `filter`,`fold`, and to explicit array indexing.

However, even the experienced programmer might have difficulties understanding that in fact, line 17 implements a prefix-sum

[1] Since `lbody` is in itself a `map`, line 27 exhibits the composition of two `map`, which, if merged, would improve the parallelism degree from n to $n \times d$.

computation[2], i.e., the `scan` at line 29. While the destructive update to `ret(j,i)` optimizes the sequential execution time, we note that, at least to some degree, it affects readability.

There are two main impediments to proving parallelism for the outermost loop `do i`. *The first issue* refers to array `ia`: the algorithm's logic is that each iteration i works with its own (independent) set of filtered indexes, i.e., `ia` should be logically declared/allocated inside the loop. The implementation optimizes the sequential case by promoting `ia`'s declaration outside the loop.

However, this results in bogus cross-iteration read-after-write RAW, write-after-read WAR and write-after-write WAW dependencies. To enable parallelism, one has to prove the validity of the reverse transformation, known as privatization, which reduces to proving that every read from `ia` is covered by a write to `ia` from the same iteration. A programmer might observe that loop `do k` at line 13 iterates precisely on the set of values computed by loop `do k` at line 6. However, most compiler solutions [22, 40] cannot establish this property, as their dependency analysis is restricted to cases where the array subscript can be expressed as a closed-form, typically affine, formula in the loop indexes. In our case, the conditional incrementation of `len` at line 8 does not satisfy this requirement.

The second issue is even more discouraging: the prefix-sum pattern of line 17 appears as a cross-iteration dependency of constant distance 1, which forms a dependency cycle that cannot be easily broken. Furthermore, prefix-sum can be written imperatively in a number of ways, and we are not aware of any compiler technique that would effectively identify/parallelize this pattern. In contrast, parallel reduction is effectively supported by pattern-matching techniques[3] [9, 29].

1.3 Performance Perspective

The previous section hinted that it is significantly more difficult to uncover parallelism from an imperative program, than it is to optimize a nearly-parallel functional version via imperative-like optimizations. This section outlines several such optimizations.

Throughout the paper, we denote by $\odot$ a binary-associative operator with neutral element $\mathtt{e}_\odot$, and write $(\mathtt{red}\ \odot)$ instead of $(\mathtt{fold}\ \odot\ \mathtt{e}_\odot)$. Furthermore, we use two common helper functions: $\mathtt{dist}_p :: [\mathtt{a}] \to [[\mathtt{a}]]$ to split the input list into a list of `p` lists of nearly equal lengths, and $\mathtt{tile}_t :: [\mathtt{a}] \to [[\mathtt{a}]]$ which chunks the list into a list of lists containing each roughly `t` elements.

Space-Reuse of Map-Reduce Functions. It is well known that $(\mathtt{red}\ \odot)\ .\ (\mathtt{map}\ \mathtt{f})$ can be formally transformed, via list homomorphism (LH) promotion lemma [4], to its equivalent form:

$$(\mathtt{red}\ \odot).(\mathtt{map}\ \mathtt{f}) \equiv (\mathtt{red}\ \odot).(\mathtt{map}\ ((\mathtt{red}\ \odot).(\mathtt{map}\ \mathtt{f}))).\mathtt{dist}_p \quad (1)$$

i.e., the input list is split into number-of-processor lists, on which each processor performs the original computation (sequentially), and finally, the local results are reduced in parallel across processors. Note that the map-reduce in the middle does not need to instantiate the list-result of `map`, i.e., destructive updates can be used to accumulate each output of `f` to the local result. This requires a total memory space proportional to `p`, rather than `N` (the list's length).

The latter form is typically preferred on massively parallel systems to optimize the communication cost, while the former is preferred on SIMD (vector) systems, which typically exhibit a rather uniform memory and very limited per-processor resources.

GPUs are, in a sense, a mix of both: a GPU is pseudo-SIMD, but features a non-homogeneous memory, in which the local memory close to the core is several orders of magnitude faster than the global one. We identify an interesting trade-off: if the size of `f`'s output fits in the fast (local) memory, then the application becomes compute-bound rather than memory-bound. The downside is that increasing the per-core resources decreases the parallelism degree of the system, and, as such, its effectiveness at hiding various kinds of latencies. Section 3.2 explores this trade-off in detail.

Strength Reduction is a transformation that replaces an expensive operation (`*`) with a recurrence that uses a cheaper operation (`+`):

```
k = k0                                k = k0
do i = 1,N       Strength Red.      doall i = 1,N
  A[k] = ..      <------------        A[k0+2*(i-1)]=..
  k = k + 2      ------------>      enddo
enddo            Ind.Var.Subst.     k = k0 +MAX(2*N,0)
```

In the code snippet above, `k0+2*(i-1)` has been replaced with the cheaper recurrence `k=k+2`. The inverse transformation, induction variable substitution, replaces the recurrence with a closed-form formula in the loop index, and thereby enables parallelism extraction: (i) it eliminates the cross-iteration RAW dependency on `k` and allows the compiler to disprove cross-iteration WAW dependences on array `A`, i.e., $\mathtt{k0+2*(i_1-1)} = \mathtt{k0+2*(i_2-1)} \Rightarrow \mathtt{i}_1 = \mathtt{i}_2$.

Compilers typically support simple algebras that, for example, allow replacing multiplication/exponentiation with recurrent addition/multiplication formulas in the sequential case, and the reverse for the parallel case. Section 2.2 shows a more complex example of strength reduction and advocates that such invariants should be captured at language level, since they reveal a nontrivial and impactful optimization space, which is explored in Section 3.3.

Branch Divergence. Consider the simple target code `map fun`, where $\mathtt{fun\ i = if\ (test\ i)\ then\ (f_1\ i)\ else\ (f_2\ i)}$. When evaluating the application of `fun` to all elements of an array *in parallel* on a symmetric multiprocessor (SMP), an asymptotic worst-case time-cost ($\mathcal{C}$) estimate is $\mathcal{C}(\mathtt{map\,fun}) = \mathcal{C}(\mathtt{test}) + MAX(\mathcal{C}(\mathtt{f}_1),\ \mathcal{C}(\mathtt{f}_2))$.

In contrast, when the code runs on a SIMD machine, in case the `if` branch diverges for at least one core (one element of the array), the runtime effectively corresponds to all cores executing both branches, i.e., $\mathcal{C}(\mathtt{map\,fun}) = \mathcal{C}(\mathtt{test}) + \mathcal{C}(\mathtt{f}_1) + \mathcal{C}(\mathtt{f}_2)$.

Our solution is to tile the computation via LH promotion lemma:

$$\mathtt{map\,fun} \equiv (\mathtt{red}\,{+}{+})\ .\ (\mathtt{map}\ (\mathtt{map\,fun}))\ .\ \mathtt{tile}_t \qquad (2)$$

where `map fun` in the middle is intended to be executed sequentially, and to replace `map fun` with a semantics-preserving, efficient, imperative code that permutes `map`'s iteration space `1..t` such that the elements for which `test` succeeds are computed via $\mathtt{f}_1$ before the ones for which `test` fails (via $\mathtt{f}_2$).

To see how this approach optimizes divergence, consider the case in which two GPU cores process tiled lists $[1,2,2]$ and $[4,5,5]$, where `test = odd`. Without the transformation, the two cores execute different branches for each pair of elements. With the transformation, the lists are processed in the orders $[2,2,1]$ and $[4,5,5]$, and only the middle elements cause branch divergence.

This technique, described in detail in Section 3.4, is complemented by copying-in and out the tiled lists to and from fast memory in order to not introduce un-coalesced accesses.

Memory Coalescing is achieved on GPU when a group of neighboring cores (e.g., 16) access, in the same instruction, a contiguous chunk of memory (e.g. 64 bytes). Since the virtual memory is implemented interleaved on different memory banks, the whole chunk is brought to registers in one memory transfer (and in parallel with the accesses of all such groups of neighboring cores).

[2] `ret(j,i)` is locally computed at lines `12-15`; line `17` `xor`-aggregates, across same-row-position elements, the local contribution of iteration i to the "sum" of the previous `i-1` iterations, available in `ret(j,i-1)`.

[3] Roughly, array `A` forms a reduction pattern for loop L, if it is used in L only in statements of shape $\mathtt{A(e_1,..,e_n)=A(e_1,..,e_n)}\odot\mathtt{e}$, where `e` is free of `A` and $\odot$ is associative.

As explained in Section 3.5, one can transparently restructure arrays and indexes to enable coalesced accesses. For example, consider the code `map (red ⊙)::[[Int]]->[Int]`, where the input list represents a $N \times 32$ matrix and `map` is parallelized. In each instruction, each group of 16 cores accesses addresses 128 bytes apart from each other, which requires 16 memory transfers, resulting in inefficient bandwidth utilization. For example, if 16 divides N, the layout can be changed to a three-dimensional array $N/16 \times 32 \times 16$, and an access to row x and column y is mapped to index (x `‘div‘` 16, y, x `‘mod‘` 16), achieving coalesced accesses.

1.4 Main Contributions

We consider the following main contributions of this paper:

- a side-by-side comparison of functional *vs* imperative code patterns that provides evidence that parallelism is easier to recognize in the former style, while the latter style, at least in part, requires the compiler to reverse-engineer sequential optimizations which are second nature to the imperative programmer,
- four compiler optimizations that (i) take advantage of the expressive, map-reduce functional style to derive simple yet powerful program transformations, and (ii) seem well-suited for integration into the repertoire of a GPU-optimizing compiler,
- an empirical evaluation on a relatively-complex financial kernel that demonstrates (i) a rich trade-off/optimization space, and (ii) that the proposed optimizations have significant impact,
- from a pragmatic perspective, we demonstrate speedups as high as $68\times$ and on average $43\times$ against the sequential CPU execution on a commodity, mobile GPU, and $\sim$ 10 times that on a mid-range GPU.

2. Generic Pricing Algorithm and Invariants

Section 2.1 provides the algorithmic background of our generic-pricing software, explaining the Monte Carlo method used and its salient configuration data. We then illustrate how the computational steps in the algorithm translate to a composition of functional basic blocks that expose the inherent parallelism of the algorithm as instances of well-known higher-order functions. The interested reader is advised to refer to Hull (2009) [26] and Glasserman (2004) [19] for a more detailed description of the financial model and the employment of Monte Carlo methods in finance, respectively.

Section 2.2 advocates the need to express high-level invariants at language level: In the context of the Sobol quasi-random-number generator, we identify a strength-reduction pattern and demonstrate that (i) its specification can trigger important performance gains, but (ii) the latter should be compiler's responsibility.

2.1 Functional Basic Blocks and Financial Semantics

Financial institutions play a major role in providing stability to economic activities by reallocating capital across economic sectors. Such crucial function is performed by insuring and re-balancing risks deriving from foreseeable future scenarios, which are appreciated by quantifying at present time the economic impact of future events. This is performed by means of (i) a probabilistic description of these (at now unknown) events, and (ii) a method to quantify at present time their economic value. Risk management is then performed by allocating capital according to the present value of the available opportunities for investment, while at the same time insuring against outcomes that would invalidate the strategy itself.

With respect to this, options are among the most exchanged contracts between two financial actors. The future value of these contracts is typically formulated via mathematical functions over a set of market assets, named underlyings. At a set of exercise dates

```
mc_pricing n = sum gains / fromIntegral n
  where c     = init_pricing n
        gains = map ( payoff c           -- ℝ^(u×d) → ℝ
                    . black_scholes c    -- ℝ^(u×d) → ℝ^(u×d)
                    . brownian_bridge c  -- ℝ^(u·d)  → ℝ^(u×d)
                    . gaussian           -- [0,1)^(u·d)  →  ℝ^(u·d)
                    . sobolInd c         --  Int  → [0,1)^(u·d)
                    ) [0..n-1]
```

Figure 2. Functional Basic Blocks and Types

specified in the option contract, the insuring actor will reward the option holder with a payoff typically depending on the future values assumed by the underlyings. The vanilla European option on single underlying is an example of contract with one exercise date, where the payoff at expiration time (maturity) will be the difference, if positive, between the future value of the underlying and a threshold (strike) set at contract issuing. Options with multiple exercise dates may also force the holder to exercise the contract before maturity, in case the underlyings crossed specific barrier levels before one of the specified dates.

We have tested our pricing engine in particular with three commonly exchanged contract types: a European vanilla option over an index, a discrete barrier option over multiple underlyings where a fixed payoff is function of the trigger date, and a barrier option monitored daily with payoff conditioned on the barrier event and the market values of the underlyings at exercise time. The underlyings of the latter contracts are the area indexes Dj Euro Stoxx 50, Nikkei 225, and S&P 500, while the European option is based on the sole Dj Euro Stoxx 50. The number of monitored dates is one for the European case, and 5 and 367 for the two barrier contracts.

The present value of these contracts, under assumption of ideal markets, is function of the payoff at exercise date. Two key elements are therefore necessary in our engine: (i) a stochastic description of the underlyings allowing to explore the space of possible trigger events and payoff values at the specified exercise dates, and (ii) a technique to efficiently estimate the expected payoff. In this context, we employ the quasi-random population Monte Carlo method, where samples are initially drawn from an equi-probable, homogeneous distribution, and are later mapped to the probability distributions chosen to model the underlyings. For this purpose, we employ the Sobol multidimensional quasi-random algorithm [10]. This method is known to provide efficient entropy, leading to homogeneous coverage of the sampling space, and thus to a good estimation of the possible payoff values. Additionally, it exhibits a strength-reduction invariant that enables an efficient parallel implementation providing identical semantics to the sequential algorithm. In contrast, most parallel techniques use multiple generators starting from different seeds, which leads to nonisometric sampling and introduces inaccuracies in the pricing algorithm [19].

The samples produced by the quasi-random-number generator are then employed to describe the behavior of the underlyings. From a modeling perspective, the aforementioned market indexes exhibit very good liquidity and present no discontinuities. With good approximation they can therefore be independently modeled by a composition of Brownian motions, where the increments of each underlying follow Normal distributions [6]. The values of the underlyings at the monitored dates are therefore connected by Brownian bridges [26]. Finally, correlation between the underlyings is imposed via Cholesky composition, by means of a positive-definite correlation matrix L provided as input [45].

Figure 2 shows how these algorithmic steps directly translate into a composition of essential functions. The first step (given last in the function composition) is to generate independent pseudo-random numbers for all underlyings, u, and dates, d, by means of the Sobol's quasi-random number algorithm (function `sobolInd`). The samples have support in the unit hypercube of dimension $u \cdot d$, and are therefore distributed in the interval $[0, 1)$. They are then mapped to Normally distributed values by a cumulative probability inversion based on the quantile method (function `gaussian`) [46].

The next step, `brownian_bridge`, maps the list of random numbers to Brownian bridge samples of dimension $u \cdot d$. This steps induces a dependency between the columns of the samples matrix, i.e. in the date dimension $1..d$. In the next step, `black_scholes`, the underlyings, stored in the rows of the matrix, are mapped to their individual stochastic process. Correlation is then imposed by Cholesky composition, inducing a dependency in the row dimension. Finally, the `payoff` function computes the payoff from the matrix of the monitored values of the underlyings.

The outer Monte Carlo level, given by the `map` function, repeats this procedure a number of times equal to the size of the stochastic population. First order statistics is then collected by averaging over the payoff values.

From a developer perspective, this functional outline allows to fully appreciate the composition possibilities and the reusability of this solution. In fact, the only function strictly dependent on the contract type is the `payoff` function, while all the other modules can be freely employed to price options having underlyings modeled with similar stochastic processes. Further, Figure 2 evidences the inherent possibilities for parallelism: distribution (`map`) and reduction (`sum`) are immediately evident, and the functional purity allows to easily reason about partitioning work and dependencies. The Haskell code shown here has in fact been written as a prototype for reasoning about potential parallelization strategies for a C+GPU version; while at the same time providing the basis for an optimized Haskell version for multicore platforms.

2.2 Algorithmic Invariants: Sobol sequences

Algorithm. A *Sobol sequence* [10] is an example of a *quasi-random*[4] or *low-discrepancy* 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 successively better representative uniform samples of the unit hypercube. Sobol sequences achieve a discrepancy of $O(\frac{\log^s n}{n})$, which means that there is a constant c (which may depend on s, but not n) such that

$$|\#\{x_i \mid x_i \in \prod_{j=1}^{s}[a_j, b_j) \wedge i < n\} - n\prod_{j=1}^{s}(b_j - a_j)| \leq c \log^s n$$

for all $0 \leq a_j < b_j \leq 1$.

Let us denote the canonical bit representation of non-negative integer n by $B(n)$, with B^{-1} mapping bit sequences back to numbers. The algorithm for computing a Sobol sequence for $s = 1$ starts by choosing a *primitive* polynomial $P = \sum_{i=0}^{d} a_i X^i$ of some degree d over the Galois Field $GF(2)$, with $a_0 \neq 0, a_d \neq 0$. The second step is to compute a number of *direction vectors* m_k via a recurrent formula that uses P's coefficients:

$$m_k = (\bigoplus_{i=1}^{d} a_{d-i} m_{k-i}) \oplus 2^d m_{k-d}$$

for $k \geq d$, where $m \oplus n = B^{-1}(B(m)\,\texttt{xor}\,\,B(n))$ and `xor` denotes the exclusive-or on bit sequences. The values of m_i for $0 \leq i < d$ can be chosen freely such that $2^i \leq m_i < 2^{i+1}$. In the third step, we compute *Sobol proxies* via the *independent* (as opposed to recurrent) formula

$$x'_i = \bigoplus_{j \geq 0} B(i)_j m_j$$

where $B(i)_j$ denotes the j-th bit of $B(i)$. (The 0-th bit is the least significant bit.) Finally, reading the binary representation of Sobol proxies as a fixed point number yields the Sobol number x_i:

$$x_i = \sum_{j \geq 0} B(x'_i)_j 2^{-j-1}.$$

```
-- Independent Formula
sobolInd :: Config -> Int -> [ Int ]
sobolInd c i = map xorVs (sobol_dirs c)
          where
          inds     = filter (bitSet (grayCode i)) [0 .. bitsnum-1]
          xorVs vs = fold xor 0 [ vs!i | i <- inds ]
-- Generating the first n numbers using the independent formula:
-- map (sobolInd c) [1..n]

-- Recurrent Formula INVAR: i ≥ 0 ⇒
-- sobolInd (i + 1)  ≡  sobolRec (sobolInd i) i
sobolRec :: Config -> [Int] -> Int -> [Int]
sobolRec c prev i = zipWith xor prev dirVs
    where dirVs = [ vs!bit | vs <- sobol_dirs c]
          bit   = least_sig_0bit i
-- Generating the first n numbers using the recurrent formula:
-- scan (sobolRec c) (sobolInd c 0) [1..n-1]
```

Figure 3. Sobol Generator: Independent *vs* Recurrent Formulas.

Instead of using $B(n)$ in the definition of Sobol proxies we can use the *reflected binary Gray code* $G(n)$, which can be computed by taking the exclusive or of n with itself shifted one bit to the right: $..g_2g_1g_0 = ..b_2b_1b_0 \oplus ..b_3b_2b_1$). This changes the sequence of numbers produced, but does not affect their asymptotic discrepancy. It enables the following *recurrence formula* for Sobol proxies:

$$x'_{n+1} = x'_n \oplus m_c$$

where c is the position of the least significant zero bit in $B(n)$.

A Sobol sequence for s-dimensional values can be constructed by s-ary zipping of Sobol sequences for 1-dimensional values.

Invariants. Figure 3 shows the essential parts of our Haskell implementation for s-dimensional quasi-random Sobol proxies.[5] The function `sobolInd` implements the independent formula with the optimization that n's bits set to one are filtered and the result is reduced via `xor`. The recurrent formula is implemented by the `sobolRec` function: the least significant zero bit is used to select the corresponding set of direction vectors (`dirVs`), which are `xor`ed with the corresponding entries of the previous vector (`zipWith xor prev`).

Section 1.3 has outlined an example of *strength reduction*, in which a repeated multiplication was replaced via a computationally cheaper, plus-recurrence formula. We observe that `sobolInd` and `sobolRec` match the strength reduction pattern: Computing the first n vectors via `sobolInd` is embarrassingly parallel, i.e., the `map` in Figure 3, while the strength-reduced `sobolRec` is significantly cheaper but requires a $\log n$-depth algorithm (`scan`).

The imperative Sobol code, not presented here, exhibits the patterns discussed in Section 1.2 that would preclude parallelism discovery. Another illustrative example corresponds to the Brownian-bridge implementation, shown in Figure 4: each iteration `i` reuses the space of array `wf` and accumulates the result in `res`. This space-saving technique, together with the indirect indexing makes it very

[4] The nomenclature is somewhat misleading since a quasi-random sequence is neither truly random nor pseudo-random: It makes no pretenses of being hard to predict.

[5] Our code actually computes the integers corresponding to the *reverse* bit representation of Sobol proxies, as in Sobol's original work, and the configuration parameter carries also information for the other functions involved in pricing.

```
REAL zd(u,d), wf(u,d)
DO i = 1, N ...
  DO m = 1, u
    wf( m, bb_bi(0)-1 ) = bb_sd(1) * zd(m, 1);
    DO j = 2, d
      wk = wf( m, bb_ri(j) - 1 );
      zi = zd( m, j );
      wf( m, bb_bi(j) - 1 ) = bb_rw(j) * wk + bb_sd(j) * zi
      IF (bb_li(j) - 1 .NE. -1)
        wf( m, bb_bi(j) - 1 ) += bb_lw(j) * wf( m, bb_li(j) - 1)
    ENDDO
  ENDDO
  ...    res = res + wf(..,..) ...
ENDDO
```

Figure 4. Brownian-Bridge Code Snippet

difficult to prove that each read from `wf` in iteration `i` is covered by a corresponding read to `wf` in the same iteration `i`, i.e., the loop `do i` can be parallelized by privatizing array `wf`. The functional style would likely expand array `wf` with an outermost dimension of size `N`, and express the loop as an easily-parallelizable `map-reduce` pattern, in which `map`'s function is given by the `do m` loop.

Discussion. This paper takes the perspective that the compiler should be the depositary of the knowledge of how best to optimize a program, while the user should primarily focus on the algorithmic invariants that (i) are typically beyond the compiler's analytical abilities and (ii) would enable the application of such optimizations. There are several reasons that support this view:

First, specifying such invariants requires minimal effort, e.g., `sobolInd (i+1) c` $\equiv$ `sobolRec c (sobolInd c i) i` documents the strength reduction invariant: the independent formula can be described via a recurrence.

Second, the optimization strategy is often hardware-dependent, hence it is impossible for the user to write an optimal hardware-agnostic program. For instance, `scan sobolRec` is well suited to the sequential case, while `map sobolInd` can be better on a massively parallel machine that exhibits high communication costs.

Finally, program-level transformations are often nontrivial, and at least tedious even for the experienced user to do by hand: e.g., Section 3.3 presents how to optimize both the parallelism depth and time overhead: the computation is tiled via a factor t, where the tile amortizes the cost of one `sobolInd` over $t - 1$ (fast) executions of `sobolRec`. Another good example is *flattening* [7].

3. Optimizations

This section describes in detail several compiler optimizations that had a strong impact on the pricing algorithm, and that we believe are likely to prove effective in a general context.

In Sections 3.2 and 3.3, we describe optimizations and tradeoffs related to exploiting coarse-grained parallelism and strength-reduction invariants. These are high-level transformations demonstrated using functional code snippets. Sections 3.4 and 3.5 present two lower-level optimizations, related to branch divergence and memory coalescing, that are demonstrated on a Fortran intermediate representation.

3.1 Notation and Language Assumptions

Throughout the paper, we use Haskell to illustrate the functional programming style, but disregard lazyness issues and use lists instead of vectors for the sake of clarity. As mentioned previously, we write `(red` $\odot$`)` for `(fold` $\odot$ $\mathtt{e}_{\odot}$`)` with a binary associative operator $\odot$ and neutral element $\mathtt{e}_{\odot}$, and use helper functions $\mathtt{dist}_p$ and $\mathtt{tile}_t$ to distribute lists in two ways.

When discussing the imperative programming model, we use Fortran77 uniformly, because: (i) it accurately illustrates the original `C` code of the pricing algorithm, and, if anything, (ii) it eliminates the maybe-aliasing issue, which is a major hindrance to automatic parallelization. Furthermore, (iii) a vast amount of work in autoparallelization targets Fortran77.

```
-- vt1, vt2 ∈ ℝ^{n×(u·d)}             CC t1, t2 ∈ ℝ^{u·d}
-- vt3, vt4 ∈ ℝ^{n×u×d}, vt5 ∈ ℝ^n     CC t3, t4 ∈ ℝ^{u×d}, t5 ∈ ℝ
let                                    do i = 0, n-1
  vt1 = map sobolInd  c [0..n-1]         t1 = sobolInd        c i
  vt2 = map gaussian          vt1        t2 = gaussian          t1
  vt3 = map brownian_bridge c vt2        t3 = brownian_bridge c t2
  vt4 = map black_scholes   c vt3        t4 = black_scholes   c t3
  vt5 = map payoff          c vt4        t5 = payoff          c t4
in                                       res = res + t5
  sum vt5                              enddo
-- Memory Complexity: O(n · u · d)      CC O(P · u · d),P = core num
```

Figure 5. Vectorized (Haskell) *vs* Coarse parallelism (Fortran)

As a fourth point, (iv) Fortran77 code resembles the GPU API OpenCL which we use, in that it neither supports recurrence nor dynamic allocation (static arrays only).

Another aspect to be taken into account when discussing optimizations is data locality and thread grouping on GPUs. A GPU operates in thread *blocks*, and threads are grouped to SIMD groups (so-called *warps*) executed on one SIMD unit comprising multiple cores. To simplify our argument, we consider that each SIMD unit comprises 32 hardware cores. Technically this is not correct, as a warp resides on only 8 cores, which execute four-cycle instructions and need four threads to amortize the cost, but the analogy is valid for the points we are making. A block of size B yields $B/32$ hardware threads per core, which we call "virtual cores".

3.2 Vectorized *vs* Coarse-Grained parallelism

Section 1.3 has outlined the tradeoff related to selecting one of op(at least) two possible implementations of a *map-reduce* computation. Figure 5 illustrates these two choices in the context of the generic-pricing algorithm shown in Figure 2.

The vectorized version distributes the outer `map` across each of the basic-block kernels, and reduces the result vector in parallel via the plus operator. (This transformation is the inverse of fusion and is known as loop distribution in the imperative context.)

On GPU, vectorization exhibits the advantage that each kernel requires fewer resources per virtual core than the fused version. This potentially increases the parallelism degree, which can be used for hiding latencies. In addition, vectorization enables each kernel to be further optimized, e.g. the gaussian kernel applies function `map gaussian_elem`, hence `map gaussian` exhibits nested parallelism that can be optimized via flattening to increase the parallelism degree.

The downside is that the memory space complexity is nonoptimal, i.e., proportional to `n`, because all intermediate vectors need to be instantiated. It follows that `vt1..5` have to be allocated in global storage, which is several order of magnitude slower than the fast local memory. (Note that the superior parallelism degree hides to a certain level, but typically does not eliminate memory latency, i.e., from a point on, spawning more computation will put too much pressure on the memory system.)

The coarse-grained version is obtained via the transformation:
`(red` $\odot$`).(map f)` $\equiv$ `(red` $\odot$`).(map ((red` $\odot$`).(map f))).`$\mathtt{dist}_p$
that distributes the input list among processors, performs the original computation `f` sequentially on all processors and pre-reduces the local results in parallel. Space consumption is optimized via privatization: `t1..t5` are allocated per virtual-core, and memory is reused via destructive updates for both the privatized variables and the (accumulated) result `res`. Note that (i) `res` needs to be repli-

cated for each sub-list before a final (parallel) reduction, and (ii) the iteration scheduling policy, i.e., the list distribution, is omitted in Figure 5, since it is handled automatically by GPU's programming interface (OpenCL compiler). The main advantage of the coarse-grained version is that the memory consumption is (asymptotically) optimal: its size is proportional to the number of virtual cores rather than to the data size. When all local variables fit in the fast memory this leads to a computational, rather than memory bound behavior[6]. The downside is that (i) it requires more per-virtual-core resources than vectorization, hence it exhibits a lower parallelism degree, and (ii) it is not applicable when the local resources do not fit in fast memory.

The Cost Model must be able to compute a maximum size of per-virtual-core resources, as an upper limit from which on the benefits of using local memory are eliminated by the reduced parallelism degree failing to optimize other kinds of latency (e.g., cache and instruction latencies, register dependencies). An accurate model is difficult to implement because latencies are in general both program and data sensitive, e.g., global-memory latency depends on whether memory accesses are coalesced. In principle, this could be addressed via machine-learning and/or profile-guided techniques, but that study is beyond the scope of this paper.

A simple heuristic is to define the cut point by computing the per-virtual-core resources associated to a reasonably-minimal concurrency ratio $\mathcal{CR}^{min}$. Since the technique eliminates the global-memory latency, $\mathcal{CR}^{min}$ is related to arithmetic latency, which, on our GPU hardware requires a ratio of virtual to hardware cores between 9 and 18, depending on the existence of register dependencies. We choose $\mathcal{CR}^{min} = (9+18)/2 = 14$, and compute the associated per-virtual-core resources as $\mathcal{R}_{th} = \mathcal{M}^{sm}_{fast}/(\mathcal{CR}^{min} \cdot 32)$, where $\mathcal{M}^{sm}_{fast}$ and the denominator denote the fast-memory size and the number of virtual cores per multiprocessor, respectively.

Our hardware exhibits $\mathcal{M}^{sm}_{fast} =$ `112kB`, thus $\mathcal{R}_{th}=$ `256` bytes. In our example, each virtual core (iteration) requires storage for three vectors, each of (flattened) size $u \cdot d$: the first two are necessary because some kernels cannot do the computation in-place and the third is necessary to record the previous quasi-random vector required by the strength-reduction optimization. In addition we need about 16 integers to store various scalars, such as loop vectors. It follows that the cutoff point is $u \cdot d = 16$, which is close to the optimal in our case, but warrants a systematic validation. The cost model is implemented via a runtime test, and we observe speedups as high as $1.5\times$ when the coarse-grained version is selected.

3.3 Strength Reduction

This section demonstrates how strength reduction can trigger a code transformation that combines the advantages of both independent and recurrent formula. In essence, the user-specified invariant:
`sobolInd c (i+1)` $\equiv$ `sobolRec c (sobolInd c i) i`,
allows one to formally derive that the `(i+k)`th random number, i.e., `sobolInd c (i+k)`, can be written as a reduction of the previous `k-1` numbers: `fold (sobolRec c) (sobolInd c i) [i..i+k-1]`, and similarly, the `i`th`..(i+k)`th random numbers can be computed as a prefix sum: `scanl (sobolRec c) (sobolInd c i) [i..i+k-1]`. This is synthesized in Figure 6 by the `sobolRecMap` function that computes the (consecutive) samples index from `l` to `u`.

The idea is that tiling a `map` computation would allow to use `sobolRecMap` to efficiently (sequentially) compute tile-size consecutive random numbers, where tiles are computed in parallel. More formally, on the domain of lists holding consecutive numbers, one can derive that `map (sobolInd c)` is equivalent to `(red++).(map (map (sobolInd c))).tile`$_t$. The last step is to replace `map (sobolInd c)` with its equivalent, but more efficient form `sobolRecMap`, yielding: `(red++).(map (sobolRecMap c)).tile`$_t$.

```
-- USER SPECIFICATION
sobolInd :: Config -> Int -> [ Int ]

-- Recurrent Formula INVAR: i ≥ 0 ⇒
-- sobolInd c (i + 1) ≡ sobolRec c (sobolInd c i) i
sobolRec :: Config -> [Int] -> Int -> [Int]
sobolRec Config{..} prev i = ...

-- COMPILER GENERATED CODE
sobolRecMap conf (l,u) = scanl (sobolRec conf) fst [l..u-1]
    where fst = sobolInd conf l

tile_segm :: ((Int,Int)->[a]) -> Int -> Int -> Int -> [a]
tile_segm fun l u t = red (++) [] (map fun iv)
    where divides = (u-l+1) `mod` t == 0
          last    = if (divides) then [] else [u]
          iv      = zip [l,l+t..] ([l+t-1, l+2*t-1 .. u] ++ last)
-- COMPILER TRANSFORMS map (SobolInd conf) [n..m] TO:
sobolGen conf n m = case (cost_model conf) of
    1 -> tile_segm (sobolRecMap conf) n m tile
    2 -> map (sobolInd conf) [n..m]
    3 -> sobolRecMap conf (n,m)
```

Figure 6. Sobol Generator: Independent *vs* Recurrent Formulas.

In Figure 6 we use `tile_segm` to implement tiling, with the difference that we encode a list of consecutive numbers via a pair (l, u) denoting the lower and upper bound of the set, hence `tile_segm` returns a list of such lower-upper bound pairs. Finally, `sobolGen` selects, based on a cost model, one of the (at least) three ways to compute the `n`th to `m`th random numbers.

The Cost Model needs to select between the independent I^f, recurrent R^f and tiled T^f formulas. For the sequential execution, R^f is the most efficient. For the parallel case, we first compare I^f and R^f. Computing N elements with I^f and R^f exhibits depths (i.e., asymptotic runtime) C_{If} and $log(N) \cdot C_{Rf}$, where C_{If} and C_{Rf} are the (average) costs of one execution of I^f and R^f, respectively. It follows that I^f prevails when $log(N) > C_{If}/C_{Rf}$. On GPU, this means that I^f is superior in most cases of practical interest, because N is typically large.

Finally, to compare T^f and I^f in the parallel case, one has to model the tradeoff between the cheaper computational cost of T^f and the negative impact various tile sizes may have on the parallelism degree, and thus on the effectiveness with which latency is hidden. (A detailed exploration is beyond the scope of this paper.)

A simple model that works well on our case study and may prove effective in practice is to compute a maximal tiling size t_{max} such that it still allows for a (fixed) parallelism degree CR^{fix}, high enough to hide all latencies. For example, we pick the virtual-to-hardware-core ratio between extreme values 18 and 64 for compute and memory bound kernels, respectively.

For a input size N, we compute $t_{max} \geq 1$ as the closest power of two less or equal to N/CR^{fix}, and bound it from above via a convenient value, e.g., 128. In essence, we have circumvented the difficult problem of modeling the relation between tile sizes and latency hiding, by computing the maximal tile size that would not negatively impact on T^f. One can observe now that T^f is always superior to I^f (i.e., in terms of the work to compute N elements): $N * C_{If} \geq (C_{If} + (T-1) \cdot C_{Rf}) \cdot N/T \Leftrightarrow C_{If} \geq C_{Rf}$.

Strength reduction exhibits speed-ups as high as $3\times$, and allows an efficient Sobol implementation that computes the same result as the sequential version, modulo float associativity issues.

[6] Our example encodes the input list via an affine formula on the loop index, thus eliminating global-memory latency, but this does not hold in general.

```
CC Tiled Code; TILE | N
DO i = 1, N, TILE
  DO j = i, i+TILE-1
    IF (cond(ginp(j))) THEN
      gout(j) = fun1(ginp(j))
    ELSE
      m = ginp(j) * ginp(j)
      IF ( cond(m) ) THEN
        gout(j) = fun2(m)
      ELSE
        gout(j) = fun3(m)
ENDIF ENDIF ENDDO ENDDO

CC Inspector-Executor Code
CC initially σ=[1..TILE]
PRIVATE i,s1,s2,inp,out,σ
DO i = 1, N, TILE
  inp[1:TILE]=ginp[i:i+TILE-1]
  s1 = itPerm(id,σ,inp,TILE)
  DO j = 1, s1
    out(σ(j))=fun1(inp(σ(j)))
  ENDDO
  s2 = itPerm( sq, σ(s1+1),
               inp, TILE-s1 )
  DO j = s1+1, s1+s2
    m = inp(σ(j)) * inp(σ(j))
    out(σ(j)) = fun2( m )
  ENDDO
  DO j = s1+s2+1, TILE
    m = inp(σ(j)) * inp(σ(j))
    out(σ(j)) = fun3( m )
  ENDDO
  gout[i:i+TILE-1]=out[1:TILE]
ENDDO
```

```
CC Clone: identity (no cloning)
FUNCTION id(x)
  id = x
END
CC Clone: square
FUNCTION sq(x)
  sq = x * x
END

CC Inspector Code: computes
CC an iter-space permutation
CC that groups the true and
CC false iterations together
FUNCTION itPerm(σ, inp, size,
                cloned_code)
  INTEGER beg,end,size,σ(size)
  beg = 1
  end = size
  DO j = 1, size-1
    m = cloned_code( inp(σ(j)) )
    IF( cond( m ) ) THEN
      beg = beg + 1
    ELSE
      tmp    = σ(beg)
      σ(beg) = σ(end)
      σ(end) = tmp
      end = end - 1
    ENDIF
  ENDDO
  if( cond( inp(size) ) )
    beg = beg + 1
  itPerm = beg - 1
END
```

Figure 7. Branch Divergence Example

3.4 Branch-Divergence Optimization

Intuition. On SIMD hardware, branches that are not taken in the same direction by all cores exhibit a runtime that is as if each core executes both targets of the branch. This section proposes an inspector-executor [43] approach to alleviate this overhead: (i) the (parallel) loop is tiled, then (ii) the inspector computes a permutation of the iteration space of a tile that groups iterations corresponding to the `true` (`false`) branches together, and, finally, (iii) the executor processes the tile in the new (permuted) order. As outlined in introductory Section 1.3, organizing the (sequential) execution of a tile in this way minimizes the branch divergence across different tiles, which are processed in parallel (SIMD).

Consider the Haskell code `map f ginp`, where `f` is defined as:

```
f a =if (cond a) then (fun1 a)
     else let m = a * a
          in if(cond m) then (fun2 m) else (fun3 m)
```

The top-left part of Figure 7 shows the Fortran version of this code, where the outer loop has been tiled, and, for simplicity we assume that `TILE` divides `N`. The bottom-right part of Figure 7 shows the inspector, `itPerm`, associated to one branch target. The inspector executes the slice of the original code, i.e., `cloned_code`, that is necessary to find the direction taken by the original branch, and replaces the bodies of the branch with code that aligns the indexes of `true/false` iterations contiguously in the first/last part of σ, respectively. Finally, the split index is returned. Note that the input σ is not required to be ordered, any input permutation of the iteration space will be transformed in a permutation that groups the `true` and `false` iterations contiguously, hence σ, once initialized, does not need to be reset in the program.

The bottom-left part of Figure 7 shows the transformed code: The global-memory input associated to the tile is first copied to private space `inp`. Then, inspector `itPerm` is called to compute the permutation of the iteration space. Loop `DO j = 1, s1` executes the `true` iterations of the outer `if`, and a similar loop was intermediary generated for the `false` iterations. The latter loop was recursively transformed to disambiguate its (inner) `if` branch.

This corresponds to the second call to `itPerm` on the remaining indexes σ`(s1+1..TILE)`, in which the cloned code refers to the square-root computation of `m` in the original code that is used in branch condition `cond(m)`. Finally, the loop is distributed across the `true` and `false` iterations of the inner `if`, and the result is copied out to global memory.

Implementation. We observe that the transformation is valid only on independent loops (i.e., parallel, no cross-iteration dependencies), otherwise the iteration-space permutation is not guaranteed to preserve the original program semantics.

Consider the case when an independent loop contains only *one* outermost `if` branch. To apply the transformation: First, inline the code after the `if-then-else` construct inside each branch, or separate that code via loop-distribution to form another loop. Second, extract the inspector by computing the transitive closure of loop statements necessary to compute the branch condition, and by inserting the code that computes the permutation. Third, generate the (distributed) loops corresponding to the `true/false` iterations by cloning the loop, replacing the `if-then-else` construct with the body of the `true/false` branch, substituting the loop index `j` with σ`(j)`, and simplifying, e.g., dead-code elimination. The procedure can be repeated to optimize inner branches in the two formed loops, where each loop further refines its iteration space recorded in its corresponding (contiguous) part of σ.

If the independent loop contains two branches at the same level, then one can distribute the loop around the two branches and apply the procedure for each branch, i.e., `map` $(\mathtt{f}_1 . \mathtt{f}_2)$ can be rewritten as $(\mathtt{map}\ \mathtt{f}_1).(\mathtt{map}\ \mathtt{f}_2)$, and the `if` branches of $\mathtt{f}_1$ and $\mathtt{f}_2$ can be treated individually. Furthermore, the transformation can be applied uniformly via a top-down traversal of the control-flow (and call) graph of the original loop, where each `if`-branch target is transformed in the context of its enclosing loop.

Finally, we note that the iteration-space permutation may introduce non-coalesced accesses at the global-memory level. The copy-in/out to and from private (fast) storage technique is aimed at fixing this undesired behavior.

Cost Model. One can observe that optimizing branch divergence exhibits both fast-memory and instructional overhead: The memory overhead is related to the size of the tile, which typically dictates the size of the private input and output buffers, and the size of σ. Splitting the computation into an inspector-executor fashion may introduce instructional overhead because both the `if` condition and the `if` body may be data-dependent on the same statements, e.g., the statement that computes `m` in Figure 7. Enabling transformations, such as loop distribution, may also require either statement cloning or array expansion to fix potential data-dependencies between the two distributed loops.

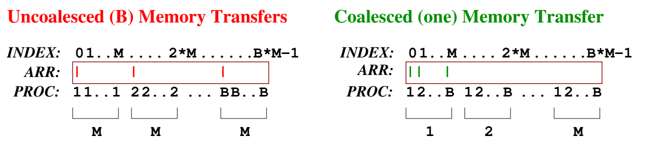
To determine the profitability of this transformation, static analysis should first identify good branch candidates, i.e., `if` statements that exhibit high computational granularity for at least one of their `true` and `false` branches (`fun1` and `fun2`). Then, similar to Section 6, the maximal tile size can be computed so that the associated fast-memory overhead does not significantly affect latency hiding.

To improve precision, runtime profiling can be used to measure the divergence ratio and to what degree the transformation would reduce divergence. Finally, the instructional overhead should be taken into account to determine whether this optimization is profitable for the target branch. With our case study, this optimization exhibits speedups (slowdowns) as high (low) as $1.3\times$ ($0.95\times$).

3.5 Memory-Coalescing Optimization

```
CC Uncoalesced Layout
DOALL i = 1, N
  DO j = 1, M
    ... ARR(σ(j), i) ...
  ENDDO
ENDDOALL
```

```
CC Coalesced Layout
DOALL i = 1, N/B
  DOALL k = 1, B
    DO j = 1, M
      ... ARR(i%B, σ(j), i/B)
ENDDO ENDDOALL ENDDOALL
```



Uncoalesced (B) Memory Transfers
INDEX: 01..M.... 2*M......B*M-1
ARR:
PROC: 11..1 22..2 ... BB..B
M M M
Coalesced (one) Memory Transfer
INDEX: 01..M.... 2*M......B*M-1
ARR:
PROC: 12..B 12..B ... 12..B
1 2 M

Figure 8. Memory-Coalescing Code Example

This section presents a transformation that fixes potential uncoalesced accesses of a map construct, such as `map fun inp`, where the elements of `inp` are arrays of similar dimensionality.

One can observe that since `map` hides the iteration space, any array indexing inside `fun` would likely be invariant to the loop that implements `map`. For example, in the left side of Figure 8, the `DOALL i` loop corresponds to the original `map`, and the `DO j` loop corresponds to `fun`, which processes an (inner) integer array of dimension `M`, indexed by σ`(j)`. Executing the `i` loop on GPU leads to the access pattern depicted in the left-bottom part of Figure 8, in which `B` cores in a SIMD group access in one instruction elements that are `4*M` bytes apart from each other, where we have assumed for simplicity $\sigma \equiv$ `id`.

We fix this behavior by reshaping uniformly such arrays via transformation $\mathbb{T}$`([x,y]) = [x/B,y,x%B]`, in which `x` and `y` correspond to the row and column index in the original matrix (note that Fortran uses column order, hence we would write `ARR(y,x)`). Since `B` is a power of two the new index is computed using fast arithmetic.

In essence, we have trimmed the outermost dimension and added an innermost (row) dimension of size `B`, the size of the SIMD group, such that one SIMD instruction exhibits coalesced access. The top-right part of Figure 8 shows the transformed code, where we made explicit the SIMD grouping via the `DOALL k` loop, while the outer `DOALL i` loop expresses the parallelism among SIMD groups. The bottom-right part of Figure 8 demonstrates that after transformation `B` consecutive cores access contiguous locations.

We observe that this transformation is effective for arrays of any dimensions, as long as the internal indexing is `map`-loop invariant. For example, assuming that the Brownian-bridge code of Figure 4 is written in map-reduce style, i.e., array expansion is applied to `wf` and `zd`, this transformation results in coalesced accesses for arrays `wf` and `zd`, despite the indirect indexing exhibited on the dates (`d`) dimension. Finally, assuming that all computational-intensive kernels are executed on GPU, it is beneficial to reshape all relevant arrays in this fashion, since the potential overheads of the CPU-executed code are in this case negligible.

We conclude by observing that this technique (i) transparently solves any uncoalesced accesses introduced by other compiler optimizations such as tiling, and (ii) yields speed-ups as high as $25\times$.

4. Experimental Results

Experimental Setup. We study the impact of our optimizations on a commodity, mobile system equipped with `8 GB` global memory, a four-core `Intel i7-2820QM@2.30GHz`, and a NVIDIA `Quadro 2000M` GPU that exhibits `192` CUDA cores running at `1100MHz`, `2 GB` global memory, `48 kB` local memory, `64 kB` constant memory, and `32768` registers per block. We compile (i) the sequential-CPU kernel with the `gcc` compiler version `4.4.3`, compiler option `-O3`, and (ii) a very similar version of the CPU code with NVIDIA's `nvcc` compiler for `OpenCL` version `4.1` with the default compiler options. Reported speed-ups were averaged among three independent runs.

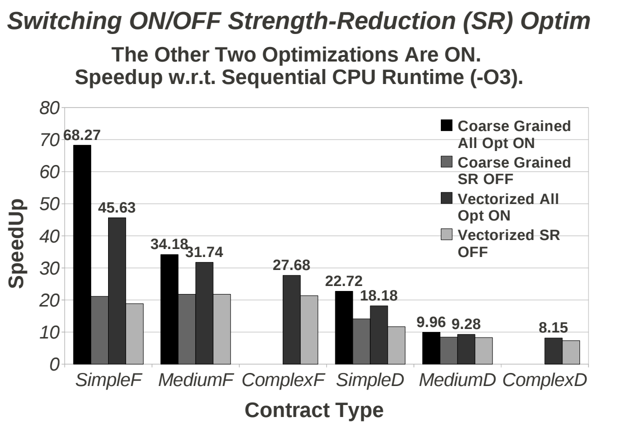


Figure 9. Impact of Strength-Reduction Optimization.

We estimate three contracts that have been described in Section 2.1: (i) an European option, named `Simple`, (ii) a discrete barrier option, named `Medium`, and (iii) a daily-monitored barrier option, named `Complex`. These contracts are written in terms of a number of underlyings u and dates d: 1×1, 3×5 and 3×367, respectively. This amounts to very different runtime behavior, since u and d dictate (i) the amount of data processed per iteration and (ii) the weight each basic-block kernel has in the overall computation.

In addition, we estimate the contracts with both single precision, e.g. `SimpleF`, and double precision, e.g., `SimpleD` floating points. From a compute perspective this accentuates the different runtime behavior, as `double` are more expensive than `float` operations (and require twice the space). From a financial perspective we note that our parallel-`double` (`float`) result is identical to the sequential one except for the last two digits (the first two digits after the comma). This is important because other solutions, rather than preserving the semantics of the sequential-random-number generator as we do, use a different seed per core. The latter may lead (i) to a larger error that might require the use of the significantly-more-expensive `double` (rather than `float`) and (ii) to additional algorithmic complications.

We show speed-ups as high as $68\times$ and $23\times$, and on average $43\times$ and $14\times$ with respect to the sequential CPU execution. On a mid-range GPU we have recently gained access initial results indicate speed-ups of up $\sim$ 500 times.[7]

Vectorization *vs* Coarse-Grained. The main optimization *choice* the compiler has to make is whether to chose coarse-grained parallelism over vectorization, as described in Section 3.2. Figure 9, in which the reader should ignore for the moment the `SR OFF` bars, demonstrates the tradeoff: The coarse-grained version on `Simple` contract exhibits a small $u \cdot d$ value (1 `float/double`), which results in (all) data fitting well in fast memory, while still allowing a good parallelism degree. It follows that the coarse-grained `SimpleF/D` is significantly faster than its vectorized analog.

As the per-core fast-memory consumption, i.e., $u \cdot d$, increases, latency is less efficiently hidden: (i) `MediumF/D` ($u \cdot d = 15$) is very close to the cutoff point between the two versions, and (ii) `ComplexF/D` cannot run the coarse-grained version simply because $u \cdot d = 3 \cdot 365$ does not fit in fast memory.

We remark that the cutoff point is (surprisingly) well estimated by the simple cost model of Section 3.2, and that the same prop-

[7] A thorough experimental evaluation is presently being conducted.

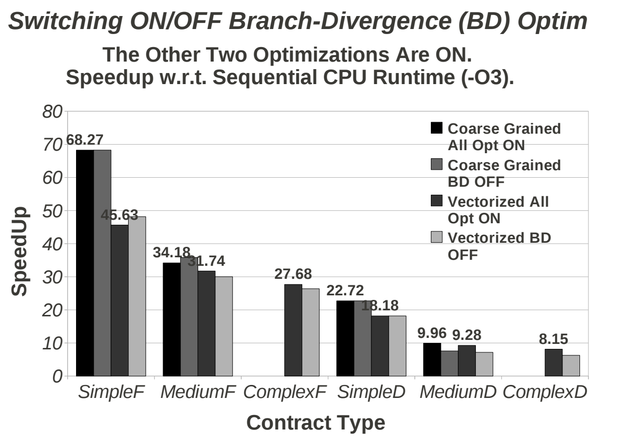


Figure 10. Impact of Branch-Divergence Optimization.

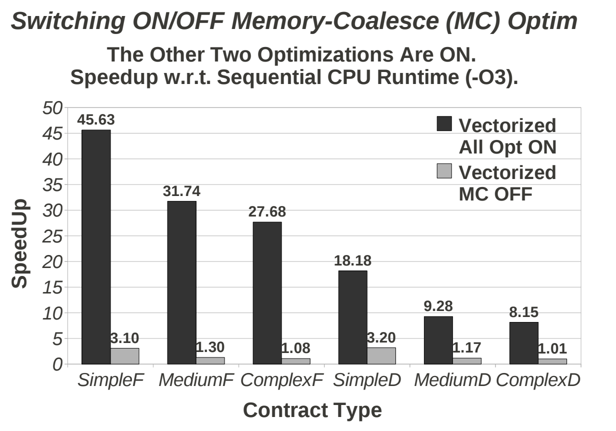


Figure 11. Impact of Memory-Coalescing Optimization.

erty holds for a top-end-gaming GPU, albeit tested (only) on the same application. At large, the top-end hardware exhibits similar behavior but superior speedups for the coarse-grained and vectorized versions of `SimpleF` ($525\times$ and $151\times$ respectively), `MediumF` ($249\times$ and $218\times$ respectively), and `ComplexF` ($154\times$ and $151\times$ respectively), all with strength-reduction optimization. The rest of this section evluates the impact of the other three optimizations for both coarse-grained and vectorized versions of the code.

Strength Reduction. The `SR OFF` bars in Figure 9 correspond to the obtained speed-up when all but the strength-reduction optimization were used. Comparing the `SR OFF` bars with their left neighbor, which correspond to the full-optimized code, one can observe speed-ups as high as $3\times$ and $2.4\times$ for `SimpleF`'s coarse-grained and vectorized code, respectively. As $u \cdot d$ increases, i.e., `Medium` and `Complex` contracts, the optimization's impact decreases because: (i) the weight of the Sobol kernel in the overall computation decreases and (ii) because the tile sizes computed by the cost model also decrease. The latter corresponds to how many times we apply the recurrence formula to amortize the more expensive independent formula, and also explains the smaller impact on the code version that uses `doubles`. For `ComplexF/D` the ratio is four and two, respectively, and the gain is smaller. We remark that the empirical data seem to validate the cost model in that the use of the recurrent formula never generates slowdowns.

Branch Divergence. The results shown in Figure 10 correspond to optimizing the divergence of the only `if` branch, located in the `gaussian` kernel, that exhibits enough computational-granularity to trigger the branch-divergence (BD) optimization. `Simple` exhibits little divergence overhead, i.e., less than 2% of the `gaussian` kernel runtime, and thus the optimization results in about 5% slowdown due to the computational and memory overheads introduced by BD. `Medium` and `Complex` exhibit roughly 61% divergence overhead at the level of the `gaussian` kernel. Examining `MediumF` we observe an interesting behavior: the coarse-grained version exhibits slowdown and the vectorized version exhibits speedup. The reason seems to be that the fast-memory overhead of BD has affected more the coarse-grained version, which was already functioning at a reduced degree of parallelism due to storing a significant amount of data in fast memory. Finally, we note that the use of `double` increases the computational granularity of the `if` branch, and, as such, the `double` version exhibits significantly better speedups (e.g., $1.3\times$), than the `float`-based one.

Memory Coalescing. Figure 11 demonstrates that achieving coalesced accesses is fundamental for extracting reasonable performance from the GPU hardware, and that the proposed transformation is effective in enabling well-coalesced accesses. We observe that in the case of `SimpleF/D` the uncoalesced accesses refer to the ones introduced by the strength-reduction optimization.

5. Related Work

A considerable amount of work has been published on parallelizing financial computations on GPUs, reporting impressive speedups (see Yoshi [47] or Giles [18], for example), or focusing on production integration in large banks' IT infrastructure [34]. Our work differs from both strands in that we aim at systematizing and eventually *automating* low-level implementation and optimization by taking a architecture-independent functional language and compilation approach.

Imperative Parallelization. Classical static dependency analysis [1, 16] examines an entire loop nest at a time and accurately models both the memory dependencies and the flow of values between every pair of read-write accesses, but the analysis is restricted to the simpler affine domain. Dependencies are represented via systems of linear (in)equations, disambiguated via Gaussian-like elimination. These solutions drive powerful code transformations to extract and optimize parallelism [39], e.g., loop distribution, interchange, skewing, tiling, etc., but they are most effective when applied to relatively small intra-procedural loop nests exhibiting simple control flow and affine accesses.

Issues become more complicated with larger loops, where symbolic constants, complex control flow, array-reshaping at call sites, quadratic array indexing, induction variables with no closed-form solutions hinder parallelism extraction [22, 37].

Various techniques have been proposed to partially address these issues: Idiom-recognition techniques [27]

disambiguate a class of subscripted subscripts and push-back arrays, such as array `ia` in Figure 1, which is indexed by the conditionally-incremented variable `len`. The weakness of such techniques is that small code perturbations may render the access pattern unrecognizable and yield very conservative results. Another direction has been to refine the dependency test to qualify some non-affine patterns: for example Range Test [8] exploits the monotonicity of polynomial indexing, and similarly, extensions of Presburger arithmetic [40] may solve a class of irregular accesses and control flow.

The next step has been to extend analysis to program level by using various set algebras to summarize array indexes interprocedurally, where loop independence is modeled via an equation on (set) summaries of shape $S = \emptyset$. The set representation has taken the form of either (i) an array abstraction [22, 37], e.g., systems of affine constraints, or (ii) a richer language [42] in which irreducible set operations are represented via explicit $\cap, -, \cup_{i=1}^{N}$ constructors. Array abstractions have been refined further to exploit (simple) control-flow predicates [32, 40] (i) to increase summary precision or (ii) to predicate optimistic results for undecidable summaries. The language-representation approach allows an accurate classification of loop independence at runtime, e.g., it can prove that array `wf` in Figure 4 is write first, hence privatizable, but the runtime cost may be prohibitive in the general case. This issue has been further addressed by a translation $\mathbb{T}$ to an equally-rich language of predicates [35], i.e., $\mathbb{T}(S) \Rightarrow S = \emptyset$, where the extracted predicates $\mathbb{T}(S)$ has been found to solve uniformly a number of difficult cases under negligible runtime overhead.

While these techniques are successful in disambiguating a large number of imperative (Fortran) loops, there are still enough embarrassingly-parallel benchmarks that are too difficult to solve statically [2]. Another avenue of investigation has been to to analyze (entirely) at runtime whether memory references produce cross-iteration dependencies. This is accomplished (i) either by optimistically running iterations in parallel together with runtime tracking/fixing of dependencies, as with thread-level speculation [15, 36], or (ii) by inspecting the traces of memory references prior to loop execution, as with the inspector/executor model [48].

Such runtime-oriented approaches typically extract maximal parallelism, albeit under significant (memory and) instructional overhead, proportional to the number of dynamic memory references. Furthermore, they typically assume an effective cache-coherent system and synchronization primitives (CAS).

A strand of research related to our branch-divergence and memory-coalescing optimizations has been the study of transformations that improve data/cache locality. Solutions include (i) parameterized tiling for the general case of convex iteration spaces [41], or (ii) the use of the garbage collector to gather profiling information that guides object-layout decomposition and object allocation [13], or (iii) inspector-executor techniques that permute the array layout to match the order in which elements are accesses at runtime [43].

The optimizations presented here, by relying on the functional map-reduce invariants, can be expressed at a higher level of abstraction and are significantly simpler than their imperative analog. For example, the memory-coalescing relies on the fact that the array-indexing used inside the `map`ped function is typically invariant across the `map`ped elements. It follows that our array-reordering transformation is likely to succeed on functional code, and is much simpler than the inspector-executor approach for optimizing locality (e.g., can be implemented entirely statically). Similarly, branch divergence optimization takes advantage of `map`'s parallelism to ensure that permuting the iteration space is a valid transformation.

Functional Parallelization.

Functional languages and their mathematical abstraction allow for more expressive algorithm implementations, in which data parallelism appears naturally by means of higher-order functions like `map`, `fold`, and `scan`, for which efficient parallel implementations are known. Consequently, research work has focused less on completely automating the process, but rather on studying in a formal manner what classes of algorithms allows asymptotically efficient (parallel) implementations.

Previous research we draw upon here is the Bird-Meertens Formalism (BMF) [4, 5]. Functions `f (x++y)=(f x)` $\odot$ `(f y)` are homomorphisms between (i) the monoid of lists with concatenation operator and empty list as neutral element, and (ii) the monoid of the result type with operator $\odot$ and neutral element `f []`, and can be rewritten in the map-reduce form which provides an efficient parallel implementation (at least when the reduction does not involve concatenation). List invariants like the promotion lemmas `(map f).(red ++)` $\equiv$ `(red ++).(map (map f))` and `(red` $\odot$`).(red ++)` $\equiv$ `(red` $\odot$`).(map (red` $\odot$`))`, can be used to transform programs to a higher degree of parallelism and load balancing ($\leftarrow$ direction), or to distribute computations to available processors for reduced communication overhead ($\rightarrow$ direction).

Other work follows this research strand and (i) studies how to extend a class of functions [14] to become list-homomorphisms (LH), or (ii) show how to use the third LH theorem [17] to formally derive the LH definition from its associated (and simpler) leftwards and rightwards forms [20, 33], or (iii) formulate a class of functions [21], such as `scan`, for which an asymptotically-optimal hypercube implementation can be formally derived, despite the fact that concatenation appears inside $\odot$, or (iv) extend the applicability of BMF theory to cover programs of more general form [24].

All these techniques rely on a functional computation language, where referential transparency and the absence of side-effects allow for vast transformations and rewriting. Such program transformations (with known operators) play a major role in compilation of functional data-parallel programs, for example in the implementation of data-parallel Haskell [11] and other variants of functional data parallelism. Other work targets GPU platforms [12] using two-stage execution techniques and JIT compilation. All approaches to data-parallel Haskell heavily rely on fusion to adjust task granularities and justify parallel overhead for the particular platform. Our work is informed by the same reasoning for the high-level optimisation (coarse-grain vs. vectorised implementation), but addresses machine characteristics more explicitly in the lower-level optimizations.

It is a general problem that functional approaches can lead to excess parallelism and too fine-grained tasks. More task-oriented parallelization techniques today follow a semi-explicit programming model of annotations (GpH [44]), or make parallelism completely explicit (Eden [28] and the Par monad [31]). Automatic parallelism in these approaches relates mainly to runtime system management, and to functional libraries that capture algorithmic patterns at a high abstraction level. Our work is more narrow in the algorithmic selection, and thereby allows for very specific optimizations.

6. Conclusions and Future Work

This paper has shown evidence that real-world financial software exhibits computationally-intensive kernels that can be expressed in a list-homomorphic, map-reduce fashion[8], and has presented and demonstrated four relatively-simple optimizations that allowed substantial speedups to be extracted from a commodity, mobile GPU hardware. While functional languages have often been considered elegant but slow, GPU's enticing parallelism and this paper's results motivates a systematic investigation of what is necessary to transparently and efficiently extract parallelism from programs written in a general-purpose functional language:

- devise the techniques that would increase the number of computationally intensive kernels that can be effectively mapped to GPU's restrictive, Fortran77-like, programming interface (i.e., no recursion, no dynamic allocation, flattened arrays),
- apply the imperative-style memory optimizations that would allow effective use of the rather-limited, per-core resources,
- develope accurate cost-models that would improve the predictability of program behavior.

[8] Note that while the Monte Carlo-based approaches are the most common, other financial modeling of the payoff, such as the ones based on partial differential equations can be expressed as map-reduce computations.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. ISBN 1-55860-286-0.

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

[3] L. Augustsson, H. Mansell, and G. Sittampalam. Paradise: A two-stage DSL embedded in Haskell. In *ICFP'08, Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 225–228, New York, 2008. ACM.

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

[5] R. S. Bird. Algebraic Identities for Program Calculation. *Computer Journal*, 32(2):122–126, 1989.

[6] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[7] G. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, 1996.

[8] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In *Procs. Int. Conf. on Supercomp*, pages 528–537, 1994.

[9] W. Blume and *et* al. Parallel Programming With Polaris. *Computer*, 29(12), 1996.

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

[11] M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel Haskell: A status report. In *DAMP'07, Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, New York, 2007. ACM.

[12] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP'11, Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, pages 3–14, New York, 2011. ACM.

[13] T. M. Chilimbi and J. R. Larus. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. In *ISMM*, 1998.

[14] M. Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In *Procs. of Parco 93*, 1993.

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

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

[17] J. Gibbons. The Third Homomorphism Theorem. *Journal Functional Programming*, 6(4):657–665, 1996.

[18] M. Giles and S. Xiaoke. Notes on using the nVidia 8800 GTX graphics card. Online report, 2007. http://people.maths.ox.ac.uk/gilesm/codes/libor/report.pdf.

[19] P. Glasserman. *Monte Carlo methods in financial engineering*. Springer, New York, 2004. ISBN 0387004513.

[20] S. Gorlatch. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *PLILP'96*, pages 274–288, 1996.

[21] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *Ann. European Conf. on Par. Proc. LNCS 1124*, pages 401–408. Springer-Verlag, 1996.

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

[23] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, London, 2000.

[24] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *Int. Workshop on Partial Eval. and Semantics-Based Prg. Manip. (PEPM*, pages 85–94, 1999.

[25] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[26] J. Hull. *Options, futures and other derivatives*. Pearson Prentice Hall, 2009.

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

[28] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3): 431–475, 2005.

[29] B. Lu and J. Mellor-Crummey. Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors. In *Int. Par. Proc. Symp.*, 1998.

[30] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell'10, Proceedings of the third ACM SIGPLAN Symposium on Haskell*, pages 67–78, New York, 2010. ACM.

[31] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, New York, 2011. ACM.

[32] S. Moon and M. W. Hall. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 84–95, 1999.

[33] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, pages 146–155, 2007.

[34] F. Nord and E. Laure. Monte carlo option pricing with graphics processing units. In *ParCo 2011, Proceedings*. IOS Press, 2012.

[35] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, 2012.

[36] C. E. Oancea, A. Mycroft, and T. Harris. A Lightweight, In-Place Model for Software Thread-Level Speculation. In *Int. Symp. on Par. Alg. Arch. (SPAA)*, pages 223–232, 2009.

[37] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 24(1): 65–109, 2002.

[38] S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00, Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 280–292, New York, 2000. ACM. (Later extended to a book chapter).

[39] L. Pouchet and et al. Loop Transformations: Convexity, Pruning and Optimization. In *Int. Conf. Princ. of Prog. Lang. (POPL)*, 2012.

[40] W. Pugh and D. Wonnacott. Constraint-Based Array Dependence Analysis. *Trans. on Prog. Lang. and Sys.*, 20(3):635–678, 1998.

[41] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized Tiled Loops for Free. In *Int. Conf. Prog. Lang. Design Implem. (PLDI)*, pages 193–202, 2007.

[42] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid analysis: Static & dynamic memory reference analysis. *Int. Journal of Par. Prog*, 31(3): 251–283, 2003.

[43] M. M. Strout. *Performance transformations for irregular applications*. PhD thesis, 2003. AAI3094622.

[44] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*, pages 78–88, New York, 1996. ACM.

[45] D. Watkins. *Fundamentals of matrix computations*. Wiley, New York, 1991. ISBN 0471614149.

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

[47] M. S. Yoshi. Graphical Asian Options. Available at SSRN, September 2009. http://ssrn.com/abstract=1473563.

[48] X. Zhuang, A. Eichenberger, Y. Luo, K. OBrien, and K. OBrien. Exploiting Parallelism with Dependence-Aware Scheduling. In *Int. Conf. Par. Arch. Compilation Tech. (PACT)*, pages 193–202, 2009.