Modular Acceleration: Tricky Cases of Functional High-Performance Computing

Troels Henriksen
Department of Computer Science,
University of Copenhagen (DIKU)
Denmark
athas@sigkill.dk

Martin Elsman
Department of Computer Science,
University of Copenhagen (DIKU)
Denmark
mael@diku.dk

Cosmin Oancea
Department of Computer Science,
University of Copenhagen (DIKU)
Denmark
cosmin.oancea@diku.dk

Abstract

This case study examines the data-parallel functional implementation of three algorithms: generation of quasi-random Sobol numbers, breadth-first search, and calibration of Heston market parameters via a least-squares procedure. We show that while all these problems permit elegant functional implementations, good performance depends on subtle issues that must be confronted in both the implementations of the algorithms themselves, as well as the compiler that is responsible for ultimately generating high-performance code. In particular, we demonstrate a modular technique for generating quasi-random Sobol numbers in an efficient manner, study the efficient implementation of an irregular graph algorithm without sacrificing parallelism, and argue for the utility of nested regular data parallelism in the context of nonlinear parameter calibration.

CCS Concepts • **Software and its engineering** → **Software performance**; **Parallel programming languages**; **Functional languages**; *Massively parallel systems*; • **Hardware** → *Emerging languages and compilers*;

Keywords parallelism, compilers, GPU

ACM Reference Format:
Troels Henriksen, Martin Elsman, and Cosmin Oancea. 2018. Modular Acceleration: Tricky Cases of Functional High-Performance Computing. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3264738.3264740

FHPC '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5813-2/18/09…$15.00
https://doi.org/10.1145/3264738.3264740

1 Introduction

Massively parallel computers, such as GPUs, are now commonplace, but the absence of convenient programming models still limit their accessibility by non-expert users. Fortunately, functional programming provides a useful vocabulary and fundamental philosophy for how to structure data-parallel applications [3]. In particular, the functional style provides a high-level description of the available parallelism, without requiring low-level details about synchronisation or evaluation order. This feature is useful in a world where machines not only get more parallel, but also more diverse. Conversely, the functional style also poses a bigger challenge to the compiler, as a looser specification requires more compiler smarts to optimize.

A rich body of related work has been aimed at investigating what is needed to support high-level expression and efficient execution of various applications on modern hardware, such as GPUs. For example, various embedded data-parallel languages have been successfully used to accelerate computational kernels of (host) conventional languages, functional or otherwise (e.g., Accelerate [8] and Lime [10]). On the downside, the embedding often places restrictions on the language and compiler design, notably they tend to not support nested parallelism. Stand-alone languages, such as NESL [2, 6] have been implemented using strategies that are general, principled, theoretically sound, and effective in many difficult cases, but they do not always support the "common case" efficiently, for example due to high communication costs.

Although the results of previous work are encouraging, the literature somewhat lacks in demonstrating the full potential of using a data-parallel language (i) for building modular applications and designing powerful high-level libraries, or (ii) for allowing a step-by-step reasoning about the major performance trade-offs, which (iii) can be modeled, empirically observed, and then optimized by reformulating the high-level implementation in a way that drives the compiler to generate the desired low-level code.

In this case study, we explore three problems to demonstrate the afore-mentioned advantages of data-parallel functional programming, as well as the requirements that a language and compiler must fulfill:

Implementation of Breadth-First Search, based on a port of a hand-written OpenCL implementation from the Rodinia [9] benchmark suite, is shown in section 2. We show how the high-level approach permits data-parallel reasoning about the implementation performance and allows algorithmic tuning of the data-parallel implementation.

Simulation of quasi-random Sobol numbers is shown in section 3, and presents the design of a convenient and modular library for efficiently generating large quantities of random numbers in a parallel and purely functional setting. Sobol numbers are particularly useful for Monte Carlo simulations, which are a core application of massive parallelism.

Heston parameter calibration, section 4, is a more complex application derived from a real financial software product, and shows the importance of handling nested parallelism. It is known that *irregular* nested parallelism is very difficult to handle efficiently, but this program shows that there is a simpler case of *regular* nested parallelism that is easier to handle, but still crucial for some applications. In particular, we argue that it is necessary to exploit two levels of parallelism that cross an abstraction boundary, and that any manual flattening by a human programmer would result in significantly less reusable code.

The source code for all three problems is publicly available at https://github.com/diku-dk/futhark-fhpc18.

2 Case Study: Rodinia Breadth-First Search

This section presents an experiment in which we start with the Rodinia (imperative) implementation of breadth-first search (BFS) as a baseline, we translate it to Futhark [15, 16, 18, 19] and successively improve it based on data-parallel (high-level) reasoning so that it efficiently covers various classes of datasets on GPUs. We believe that this experiment demonstrates that:

1. efficient high-level functional implementations can be derived even for highly-irregular parallel code, and
2. prototyping in such a data-parallel, hardware-independent language significantly enhances productivity: once a base parallel version was (correctly) translated, it took about three days to perform the work reported in this section (including deriving the four code versions, generating the datasets and analysing their performance).

Obviously, efficient low-level BFS implementations for GPU exist [26]; we emphasize that this section does *not* claim that our implementation is competitive with those. Instead, our thesis is that the Rodinia's implementation is illustrative for a lot of the GPU code developed in industry: they are reluctant in moving away from low-level GPU "assembly" (OpenCL, CUDA), but they also set hard limits on development time, which often results in code heuristics that are seriously limping from a data-parallel perspective.

2.1 Rodinia BFS Implementation

Figure 1 shows the Rodinia's BFS implementation. The program input consists of the `num_edges`, `starts_at` and `edges` arrays, which encode the graph: `num_edges` records the number of edges for each of the `n` nodes of the graph, and `starts_at[i]` records the starting index in the `edges` array where the `num_edges[i]` nodes connected to node `i` are stored. The `mask`, `visited` and `cost` arrays are initialized with **`false`**, **`false`** and `-1` values, respectively, except for the root node (at index `0`), which is initialized with **`true`**, **`true`** and `0` (not shown). The program's result is the `cost` array, which records the breadth level of each node in the graph.

```
1  do{ // sequential
2    for(t = 0; t < n; t++) { // parallel
3      if (mask[t]) {
4        mask[t] = false;
5        s = nodes[t].starts_at;
6        e = nodes[t].num_edges;
7        for(i=0; i<e; i++) { // sequential
8          int id = edges[s+i];
9          if(!visited[id]) {
10           cost[id] = cost[t] + 1;
11           updt_mask[id] = true;
12   } } } }
13
14   continue = false;
15   for(t = 0; t < n ; t++) { // parallel
16     if (updt_mask[t]) {
17       mask     [t] = true;
18       visited  [t] = true;
19       updt_mask[t] = false;
20       continue     = true;
21   } }
22 } while(continue);
```

Figure 1. Imperative code for breadth-first search.

The implementation consists of a sequentially-executed **`do-while`** loop, in which iteration numbers correspond to the breadth levels of the graph. The implementation maintains the invariants that, at the entry to iteration j: (i) all elements of `updt_mask` are **`false`**, and (ii) the `mask`'s entries are set (to **`true`**) only for the nodes corresponding to the previous breadth level ($j-1$). Each iteration of the **`do-while`** loop consists of a composition of two **`for`** loops, which are both *executed in parallel*. The first loop selects the nodes `t` on the previous breadth level (`mask[t]=true`), and then it *sequentially* traverses all the nodes `id` connected with `t`. If `id` was not visited yet, then its breadth level is set to `cost[t]+1`—note that output dependencies are possible, but the algorithm ensures that all conflicting writes are idempotent—i.e.,

they write the (same) value of the current breadth level. It is perhaps important to note that such an implementation—in which the inner loop is sequential—is not expressible or derivable by the Futhark language or compiler.

The second parallel loop updates the `mask`, `visited` and `updt_mask` arrays; if no nodes on the current breadth level are found (i.e., `updt_mask[t]` is uniformly **`false`**) then `continue` remains **`false`** and the algorithm terminates.

While the Rodinia implementation is work efficient, and holds the upper hand on some datasets, it also presents several performance inefficiencies:

1 The innermost loop (lines 7-12) is executed sequentially, which violates the depth asymptotic and results in the hardware being underutilized on the graphs in which the number of nodes (on a breadth level) is less than the hardware-supported degree of parallelism.
2 Furthermore, sequentializing the inner loop results in the access to `edges[i+s]` being non-coalesced in global memory. (In contrast, if the inner loop is parallelized than the e consecutive threads processing the node's neighbors ($i = 1 \ldots e$) would access consecutive memory locations (i.e., improved coalescing).
3 Finally, when the number of edges is highly variant, the computation is unbalanced and one can expect a significant-divergence overhead (across GPU threads).

2.2 Futhark's Scatter Data-Parallel Operator

Futhark's implementations of BFS, discussed in the next sections, rely heavily on the **`scatter`** (parallel write) operator and its fusion rules. Since these are not described elsewhere, for completeness, this section is dedicated to them.

In the source language, (**`scatter`** `x inds vals`) simply updates (in place) the elements of array `x` at the indices provided in `inds` with the values provided in `vals`; if an index falls outside the bounds of `x` then the corresponding update is ignored[1]. Its size-dependent type is given below:

$$\forall\, nm.\ *[n]\alpha \ \to\ [m]\texttt{i32} \ \to\ [m]\alpha \ \to\ *[n]\alpha$$

A uniqueness-type mechanism [19] mediates between the in-place and purely-functional semantics assumed by **`scatter`**: the star ($*$) in front of the first-parameter type indicates an unique array that will be consumed—i.e., any reference to it on any possible execution path following this program point will result in a compilation error—and the star in front of the result means that it does not alias any of the non-unique parameters (i.e., the index and value arrays).

Since **`scatter`** puts a lot of pressure on the memory system, its performance depends heavily on how aggressively the compiler can fuse the computation that produces its input indices and/or values arrays. It follows that the source language provides the user with an easy-to-understand operator, but the compiler IR supports a more complex operator that corresponds to a **`scatter-map`** composition of type:

$$\forall\, nm.\ \overline{*[n]\alpha^{q}} \ \to\ (\overline{\beta^{r}} \to \overline{(\texttt{i32}, \alpha^{q})}) \ \to\ \overline{[m]\beta^{r}} \ \to\ \overline{*[n]\alpha^{q}}$$

where the third array parameter (of element type β) is mapped by the (second) function parameter to produce the index-value pairs for the update. The type uses the notation $\overline{t^{n}}$ to denote the sequence $t_1 \ldots t_n$; this is because the source language expresses the bulk-parallel operators on array of tuples, but the compiler always transforms it to a tuple-of-arrays representation—i.e., **`zip/unzip`** are compiled away.

Figure 2 presents **`scatter`**'s rewrite rules: Rule F0 transforms the source language **`scatter`** to its compiler IR, named **`scattermap`**. Rule F1 shows the fusion of a **`map`** that produces an array that is consumed by the **`scattermap`**'s function argument (dubbed vertical fusion). This rule is valid only if the input to and the free variables of the **`map`** function do not alias the to-be-updated arrays—because the GPU does not actually executes in SIMD fashion. If one still desire fusion in this (aliasing) case, one should pass to **`scatter`** an explicit `copy` of the to-be-updated array.

Finally, rule F3 shows the fusion rule of two independent **`scattermap`**s, which is applied only when (i) the two are not in any produce-consumer relation—dubbed horizontal fusion—*and* (ii) the lengths of their mapped arrays are equal.

F0 (scatter to scattermap)

```
scatter x is vs ⇒
scattermap (x) (\i v -> (i, v)) is vs
```

F1 (vertical fusion: map into scattermap)

```
scattermap (x) f e^(n) (map g xs) e^(m)   ⇒
scattermap (x) (\y^(n)x z^(m) → f y^(n)(g x) z^(m))
           e^(n) xs e^(m)
```

(Applies only if x does not alias xs or g's freevars.)

F2 (horizontal fusion of scattermaps)

```
(scattermap (x) f e^(n), scattermap (y) g e^(m))
⇒ scattermap (x,y)
      (\v^(n) v^(m) -> (f v^(n), g v^(m))) e^(n) e^(m)
```

(Applies only if $\overline{e^{(n)}}$ have the same outer size as $\overline{e^{(m)}}$.)

Figure 2. Fusion rules for **`scattermap`**

2.3 BFS Skeleton in Futhark

Figure 3 shows the common part of all Futhark implementations of BFS. The `mask`, `visited` and `cost` arrays are initialized at lines 4-9. The remaining implementation consists of the **`while`** loop, which iterates as long as `continue` is **`true`**, and which has the semantics that the result of the loop-body expression is bound to the loop-variant variables (`cost`, `mask`, `visited`, `updt_mask`, `cont`) for the next iteration of the loop.

[1] This semantics allows "padding" the array of indices and values with "noops". It also makes **`scatter`** implicitly safe to execute, which is motivated by the fact that OpenCL lacks support for assertions.

```
1  let bfs[n][e]( starts_at: [n]i32
2               , num_edges: [n]i32
3               , edges: [e]i32 ) : [n]i32 =
4   let (mask, visited, cost) = unzip (
5     map (\i-> if i == 0
6               then (true,true,0)
7               else (false,false,-1)
8         ) (0...n-1) )
9   let updt_mask = replicate n false
10  let continue  = true
11  let (cost,_,_,_,_) =
12   loop(cost,mask,visited,updt_mask,continue)
13    while continue do
14      let (cost', mask0, updt_mask0) =
15        core(edges, starts_at, visited,
16             num_edges, mask, updt_mask, cost)
17
18      let (inds, inds0) = unzip (
19         map (\i -> if (updt_mask0[i])
20                    then (i,0) else (-1,-1)
21             ) (0...n-1) )
22      let visited' = scatter visited inds
23                           (replicate n true)
24      let mask'    = scatter mask0 inds
25                           (replicate n true)
26      let updt_mask'=scatter (copy updt_mask0)
27                           inds (replicate n false)
28      let cs = scatter (copy [false])
29                       inds0 (replicate n true)
30      in(cost',mask',visited',updt_mask',cs[0])
31  in cost
```

Figure 3. Data-Parallel BFS in Futhark: various strategies can be used to implement the core function.

The call to the core function at lines 4-9 implements the first parallel **for** loop of the Rodinia implementation, and following sections will explore its optimization space.

The remaining loop-body (lines 18-29) correspond to the functionality of the second Rodinia parallel for loop, which updates in place the mask, visited and updt_mask arrays and computes the continuation condition, only that the Futhark implementation seemingly computes them with four independent **scatter** and a **map** parallel operations.

This would be very expensive if executed as such because it will require (i) manifesting in memory the inds and inds0 arrays produced by **map**, and then (ii) repeated traversals of them for each **scatter**, in addition to (iii) manifestation of the arrays produced by **replicate**.

However, the rules presented in Figure 2 allow the compiler to (automatically) fuse all five parallel operators into one **scattermap**, which closely resembles the second **for** loop of the Rodinia implementation (which is quasi optimal). The difference is the copy updt_mask0 expression at line 26, which is pure overhead and which is required in order to

```
1  let core [n][e]
2       ( edges  : [e]i32,  starts_at: [n]i32
3       , visited: [n]bool, num_edges: [n]i32
4       , mask   :*[n]bool, updt_mask:*[n]bool
5       , cost: *[n]i32 ) :
6       ( *[n]i32, *[n]bool, *[n]bool ) =
7    let inds = filter (\i-> mask[i]) (0...n-1)
8    let act_costs = map (\t -> cost[t]) inds
9    let n_inds = length inds
10   let mask' = scatter mask inds
11                   (replicate n_inds false)
12   let e_max = i32.maximum num_edges
13   let flat_len = e_max * n_inds
14   let (chg_ids, chg_costs) = unzip ( map
15     (\ii ->
16         let (row,col) = (ii/e_max, ii%e_max)
17         let t         = inds[row]
18         let n_edges   = num_edges[t]
19         in if col < n_edges
20            then let ii = col + starts_at[t]
21                 let id = edges[ii] in
22                 if  visited[id] then (-1, -1)
23                 else (id, act_costs[row] + 1)
24            else (-1, -1)
25     ) (0...flat_len-1) )
26   let cost' = scatter cost chg_ids chg_costs
27   let updt_mask' = scatter updt_mask chg_ids
28                      (replicate flat_len true)
29   in (cost', mask', updt_mask')
```

Figure 4. Implementation of core using aggressive padding.

enable the F1 fusion rule—because updt_mask0 is used inside the to-be-fused **map** at line 19. Finally, the **scatter** at line 28, which computes the stopping condition, is semantically a reduce with the logical-or operator. Writing it as a reduction is inefficient because it requires a second pass over updt_mask0 (since scatter-reduce fusion is not supported).

2.4 Version 1: Aggressive Padding

Figure 4 shows the first version of the Futhark code—corresponding to Rodinia's first loop between lines 2-12 in Figure 1—but which exploits both levels of parallelism.

First, the node indices of the previous breadth level (active nodes) together with their costs (levels) are selected by the **filter** and **map** operations at lines 7 and 8, then their mask is reset at line 10. Then the **map** at lines 14-25 computes the indices id and cost values that need to be updated, while the update to the cost and updt_mask arrays are performed by the **scatter** operations at lines 26 and 27.

The **map** and the two **scatter** operations are fused into one **scattermap** construct, which would correspond to the parallel processing of all the edges of all the *active* nodes (with filtered indices in inds)—this is different from Rodinia

which traverses all graph nodes. This **map** operation uses aggressive padding: its length (`flat_len`) is the product of the number of active nodes with the maximal number of edges per node (across all graph nodes). The latter is computed at line 12, but the computation is invariant to the enclosing **while** loop, and, as such, it is hoisted out by the compiler (and is performed exactly once).

This implementation is competitive with the Rodinia's on most datasets that we have tried (curious cats can peak ahead at table 1), but the aggressive padding does not respect the work asymptotic. For example, if the count of active nodes is q and one active node has $q-1$ neighbors and the remaining active nodes have exactly one neighbor, then the size of the map `flat_len` is $q \times (q-1)$ (i.e., $O(q^2)$) rather than the work-efficient size $2 \times (q-1)$ (i.e., $O(q)$). In such skewed cases—in which the graph exhibits a very small number of highly-connected nodes—the performance suffers because, even though no array of length `flat_len` is manifested in memory, each "padded" thread will still perform two global-memory access (to `inds[row]` and `num_edges[t]`).

2.5 Version 2: Full Flattening

A flat-parallel implementation that respects both the work and depth asymptotic of the nested-parallel program can be obtained by applying Blleloch's flattening transformation [2, 4, 5, 27]. For BFS, applying this transformation would result in an implementation that corresponds to replacing the lines 12-25 in Figure 4 with the code shown in Figure 5. Our initial expectation, which was confirmed by experiments, has been that this version could be faster than the padded one on skewed datasets. However, we also expected that the additional (expensive) **scan** and **scatter** operations would hurt performance on the datasets in which the number of edges per node is constant or randomly distributed.

2.6 Versions 3 and 4: Iterative/One-Time Splitting

The pros and cons of the previous two BFS versions motivate searching for a common ground that results in decent performance for all datasets. Since flattening is likely too expensive in the "common" case, we attempt another method, which iteratively partitions the active nodes (at the previous breadth level) into a set of nodes that have the number of edges less than a constant multiple of the average edge count and the remaining nodes, which are recursively processed.

Figure 6 shows the code of the iterative-splitting method, where the lines 1-to-11 from Figure 4 should be inserted in the beginning. The partitioning of the node indices (line 14), is an expensive parallel operation, but is conditionally performed only when needed (i.e., when the dataset is skewed). However, this version requires on the critical path the computation of the average number of edges of the current active-node partition. This is implemented as a composition of a **map** and two **reduce** operations at lines 6-to-8. While these operators are efficiently fused [17, 21], thus minimizing the

```
1  let a_n_es = map (\t -> num_edges[t]) inds
2  let scan_n_es = scan (+) 0i32 a_n_es
3  let flat_len  = scan_n_es[n_inds-1]
4  let (tmp1, tmp2, tmp3) = unzip (
5      replicate flat_len (false, 0i32, 1i32) )
6  let wis = map (\i -> if i==0 then 0
7                       else scan_n_es[i-1]
8                ) (0...n_inds-1)
9  let active_flags = scatter tmp1 wis
10                      (replicate n_inds true)
11 let trk_n0 = scatter tmp2 wis (0...n_inds-1)
12 let act_st = map (\t -> starts_at[t]) inds
13 let trk_i0 = scatter tmp3 wis act_st
14 let (track_nodes, track_index) = unzip (
15     segmented_scan(\(a,b) (c,d)-> (a+c,b+d))
16                   (0,0) active_flags
17                   (zip trk_n0 trk_i0) )
18 let (chg_ids, chg_costs) = unzip ( map2
19       (\row ii ->
20            let id = edges[ii]
21            in  if visited[id]  then (-1, -1)
22                else (id, act_costs[row] + 1)
23       ) track_nodes track_index
```

Figure 5. A fully-flattened implementation is obtained by replacing lines 12-29 in Figure 4 with this figure's code.

```
1  ...
2  let continue = true
3  let (cost_res, updt_mask_res, _, _) =
4    loop (cost, updt_mask, inds, continue)
5    while continue do
6      let a_n_es= map (\t->num_edges[t]) inds
7      let max_n_edges = i32.maximum a_n_es
8      let tot_n_edges = i32.sum     a_n_es
9
10     let e_max = 3*tot_n_edges/(length inds)
11     let continue' = max_n_edges > e_max
12     let (inds_now,inds_next) =
13      if not continue' then (inds,[]) else
14       partition(\t->num_edges[t]<=e_max) inds
15
16     let flat_len = e_max* (length inds_now)
17     let (chg_ids, chg_costs) = unzip ( map
18       (\ii ->
19          let (row,col) = (ii/e_max, ii%e_max)
20          let t = inds_now[row] ...
21       ) (0...flat_len-1) )
22     let cost'= scatter cost chg_ids chg_costs
23     let updt_mask'= scatter updt_mask chg_ids
24                     (replicate flat_len true)
25     in (cost',updt_mask',inds_next,continue')
26 in (cost_res, mask', updt_mask_res)
```

Figure 6. Iterative node-splitting implementation is obtained by substituting lines 12-29 in Figure 4 with this figure's code.

number of global memory accesses and inter-thread communication, they still exhibit non-negligible overhead.

The fourth and last version (not shown) reduces the latter overhead by always splitting at most once, and by taking the split/non-split decision globally, by considering all graph nodes (rather than only the nodes at the current breadth level). The rationale for this is that (i) it is "unlikely" that the dataset is skewed on multiple levels, and that (ii) a skewed node is "likely" to exist on most breadth levels. It follows that this version executes the **`reduce-map`** composition (globally) exactly once.

2.7 Experimental Setup

The experiments are conducted on two GPU platforms: an AMD FirePro W8100 and an NVIDIA K40 with CUDA 8.0. Rodinia implementation measures only the kernel time of the two kernels presented in Figure 1 (e.g., the allocation and initialization of arrays `mask`, `visited`, `updt_mask` and `cost` is **not** measured). The Futhark runtime accounts for all overheads, except for the context creation, program compilation and host-device transfer time of the program's input and result arrays (`starts_at`, `num_edges`, `edges`, `cost`).

BFS is evaluated on six datasets:

- **D1**: $6K$ nodes, each of them having $2K$ edges ($1K = 1000$);
- **D2**: $6K$ nodes, each node has a number of edges uniformly distributed between 10 and 3990 ($2K$ on average);
- **D3**: $400K$ nodes, each of them having 30 edges;
- **D4**: $20K$ nodes, each node has a number of edges uniformly distributed between 10 and 1190 (600 on average);
- **D5**: Rodinia's `graph1MW_6` dataset, which consists of one million nodes, where the per-node number of edges is uniformly distributed between 3 and 9 (6 on average);
- **D6**: A **skewed** dataset, consisting of 65536 nodes, in which 99% of the nodes have a number of edges uniformly distributed between 4 and 60 (32 on average) and the remaining 1% of the nodes have exactly 8192 edges.

The results on the AMD and NVIDIA GPU platforms are presented in Table 1, where row **REF** of each table shows the runtime of the Rodinia OpenCL baseline, and the remaining rows (**FV1-FV4**) show the **speedups** of the four Futhark code versions in comparison to the baseline. Each test was run 10 times and the average is reported (the standard deviation was under 3%). At large, the results confirm the design intuition. In comparison to **FV1**, **REF** gains the upper hand on dataset **D5** on NVIDIA, likely because the **`filter`** operation performed by **FV1** at every breadth level is too expensive in this context. In contrast, on datasets **D1** and **D2**, **REF** under-utilizes hardware parallelism, **D4** and **D6** suffer due to high inter-thread divergence, and the slowdown on **D3** is likely caused by non-coalesced accesses—e.g., **FV1** exhibits coalesced accesses to the `edges` array because it exploits the inner parallelism. At its turn, **FV1** offers the best (Futhark) performance on datasets **D1**-**D5**, but it suffers greatly on the skewed dataset (**D6**), because its aggressive-padding strategy spawns many unhelpful threads, which still access global memory twice (each). In contrast, the flat-parallel version **FV2** is often the slowest, but it outperforms **FV1** on the skewed dataset, mainly because it respects the work-depth asymptotic.

Table 1. Speedups on the AMD and NVIDIA GPUs of the four Futhark versions (**FV1**-**FV4**) in comparison to the Rodinia baseline OpenCL implementation (**REF**) on the six datasets (**D1**-**D6**) described in text. Row **REF** displays **runtime in milliseconds**; the other rows display **speedup**.

AMD	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
REF (ms)	27ms	40ms	34ms	23ms	15ms	55ms
FV1 (×)	5.3×	7.1×	3.4×	4.3×	1.1×	2.2×
FV2 (×)	1.1×	1.6×	**.95**×	1.0×	**.44**×	3.1×
FV3 (×)	3.6×	5.5×	2.7×	3.1×	**.87**×	6.4×
FV4 (×)	4.1×	5.9×	3.1×	3.4×	1.1×	7.3×

NVIDIA	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
REF (ms)	22ms	23ms	27ms	18ms	8.2ms	40ms
FV1 (×)	4.6×	4.3×	1.9×	2.8×	**.61**×	**.94**×
FV2 (×)	**.95**×	1.0×	**.73**×	**.74**×	**.31**×	2.2×
FV3 (×)	2.8×	2.9×	1.6×	2.1×	**.54**×	4.7×
FV4 (×)	2.9×	3.0×	1.7×	2.1×	**.61**×	5.2×

FV3 seeks a common-ground that offers decent performance for all datasets: its node splitting has significant impacts on **D6**, but at the expense of loosing performance on the other datasets (because of the overhead of performing the required **`map-reduce`** operation at each breadth level). Finally, **FV4** reduces this overhead by computing the splitting once, based on global-graph information, but it does not offer any asymptotic guarantees.

3 Case Study: Sobol Sequences

In this section, we introduce the **`stream_map`** construct. The goal is to formulate our program in a way that gives the compiler freedom to exploit exactly as much parallelism as is profitable on the target hardware.

The problem we will discuss is generating n entries from a Sobol sequence [7]. Sobol sequences are quasi-random low-discrepancy sequences frequently used in Monte-Carlo algorithms and they generalize nicely to multiple dimensions. Sobol sequences are superior to traditional pseudo-random numbers for numeric integration (by Monte-Carlo algorithm). Figure 7 shows sets of points spanning the unit square chosen using traditional parallel pseudo-random techniques (top row) and Sobol sequences (bottom row). Sobol sequences simply span the space much better than their pseudo-random counterparts. In fact, it has been shown that while the value of a multi-dimensional integral for a continuous and differentiable function can be approximated with

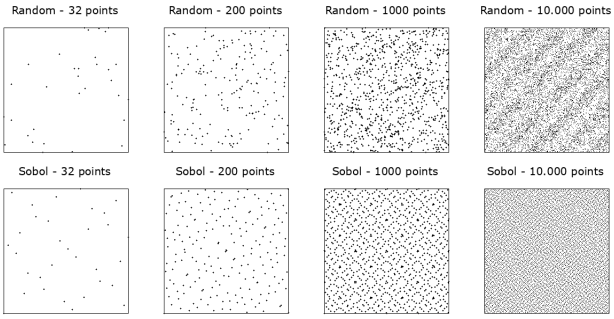

Figure 7. Points spanning the unit-square for a progressive number of points, using pseudo-random numbers (top row) and Sobol sequences (bottom row).

a convergence rate of $1/n$ using pseudo-random numbers, using Sobol sequences, the convergence rate is $1/\sqrt{n}$ [12].

For Sobol sequences, it is a requirement that the dimensionality of the sequence is given at initialisation time, as this decision affects the computation of the so-called *Sobol direction vectors*, which are specific to the choice of dimensionality. This computation is based on so-called *direction numbers*, which have been precomputed and made available in library form (as a module) [11, 20]. We will return to how parallelisation of Sobol sequences is achieved, but first, we will introduce the `Sobol` module, which is a Futhark higher-order module that takes as argument a module of type `sobol_dir` containing information about Sobol numbers and another argument specifying the dimensionality (`D`) of the generated Sobol numbers:

```
module Sobol : sobol_dir -> { val D : i32 }
               -> sobol
```

The module type `sobol` is defined as follows:

```
module type sobol = {
  val D : i32
  val sobol       : (n:i32) -> [n][D]f64
  val independent : i32 -> [D]u32
}
```

Notice that the module inherits the dimensionality `D` and that this value is referred to in the types of the embedded operations. Given a module matching the above module type, a user may easily obtain a sequence of n Sobol numbers (normalised to be floating point values in the unit-interval) by simply calling the `sobol` function. As we shall see, the underlying implementation of this function will play a series of tricks to obtain the sequence in parallel. And, moreover, the Futhark compiler will do its utmost, by fusion, to ensure that the sequence is not stored in memory at all! In general, the user can be assured that if the sequence is consumed (e.g., by a reduction operation), the sequence will not be materialized.

```
let gray_code (x: i32): i32 = (x >> 1) ^ x

let test_bit (x: i32) (ind: i32) : bool =
  (x & (1 << ind)) == (1 << ind)

let sobol_ind (dv:[L]u32) (n:i32) : u32 =
  let reldv_vals =
    map2 (\d i ->
            if test_bit (gray_code n) i then d
            else 0u32)
         dv (iota L)
  in reduce (^) 0u32 reldv_vals

let independent (n:i32) : [D]u32 =
  map (\dv -> sobol_ind dv n) dirvecs
```

Figure 8. Independent calculation of Sobol numbers.

As mentioned, the need to generate Sobol sequences efficiently comes from the need for computing multi-dimensional integrals efficiently. Such needs often arise in financial stochastic modeling such as the OptionPricing benchmark in the FinPar suite [1, 24]. The n'th Sobol number (or vector) can be computed by an independent formula, or by a cheaper (recurrent) one, which requires information about the previous Sobol number in the sequence.

An implementation of the independent formula is shown in Figure 8. The code assumes access to computed direction vectors, to appear in a variable `dirvecs` of type `[D][L]u32`, where the integer variable `L` contains the number of significant bits needed (e.g., 32).

A combined superior solution [1] that makes use of both the independent formula and the recurrent formula can be expressed elegantly with **`stream_map`** [19], whose type is:

$$\Pi n.\ (\Pi m.\ [m]\beta \ \rightarrow\ [m]\gamma) \ \rightarrow\ [n]\beta \ \rightarrow\ [n]\gamma$$

The semantics of **`stream_map`** is that its input array is partitioned into an arbitrary number of chunks, which are processed in parallel by the function argument (i.e., inter-chunk parallelism), and the result is similarly obtained by concatenating the per-chunk results. The type guarantees that the input and result chunks have the same (outer) length, but the user is responsible for ensuring the assumed property that any partitioning of the input array yields the same result.

The `sobol` function makes use of the `chunk` function, which applies the **`map`**-parallel formula once and then applies the **`scan`** formula. The code combining the recurrent formula with the independent formula is given in Figure 9. Notice that for each chunk, we first apply the function `sobol_ind` to compute the first Sobol number, then apply a combination of `map` and `scan` to compute the rest of the chunk. While

```
let index_of_least_significant_0 (x: i32): i32 =
  loop i = 0 while i < 32 && ((x>>i)&1) != 0
  do i + 1

let rec_m (i:i32) : [D]u32 =
  let bit = index_of_least_significant_0 i
  in map (\dv -> dv[bit]) dirvecs

let sobol_chunk (offs:i32) (n:i32) : [n][D]f64 =
  let sob_beg = map (\dv -> sobol_ind dv offs)
                    dirvecs
  let contrbs = map (\(k:i32): [D]u32  ->
                       if k==0 then sob_beg
                       else rec_m (k+offs-1))
                    (iota n)
  let vct_ints = scan (\x y -> map2 (^) x y)
                      (replicate D 0u32) contrbs
  in map (\xs -> map (\x -> f64.u32 x/norm) xs)
         vct_ints

let sobol (n:i32) : [n][D]f64 =
  stream_map (\[c] (xs: [c]i32): [c][D]f64 ->
                sobol_chunk xs[0] c)
             (iota n)
```

Figure 9. Chunked calculation of Sobol numbers.

`map` and `scan` are parallel operators, the compiler will sequentialise them during code generation, and instead use the chunk size to control how much parallelism to exploit.

To demonstrate the performance benefit of `stream_map` compared to explicitly parallel and sequential code, Table 2 shows runtimes for generating and summing the first thirty million 1-dimensional Sobol numbers. The results demonstrate an order-of-magnitude superiority of the combined technique. An important aspect to notice here is that while computing the sum of the 30,000,000 numbers, a **`reduce`** operation is used, which Futhark's fusion engine will fuse with the **`stream_map`** operation to achieve a streaming reduction operation, called **`stream_red`**, which avoids the intermediate allocation of the result of the call to **`stream_map`** [19].

As a final example of using the `Sobol` module, consider the code in Figure 10, which uses Sobol sequences to estimate the value of π by modeling a person throwing darts at a dartboard. The convergence effect of using Sobol sequences instead of pseudo-random numbers, for estimating the value of π, can be seen in Figure 11; with Sobol sequences the convergence towards the true value of π happens blazingly fast compared to using pseudo-random numbers.

4 Case Study: Heston Calibration

This section discusses the implementation of a calibration routine for the Hybrid Stochastic Local Volatility / Hull-White model (SLV HW model). The original formulation of

```
module sobol = Sobol sobol_dir { let D = 2 }

let main (n:i32) : f64 =
  let hits =
    map (\v -> if v[0]*v[0]+v[1]*v[1]<1.0 then 1.0
               else 0.0)
        (sobol.sobol n)
  in 4.0 * reduce (+) 0.0 hits / f64.i32 n
```

Figure 10. Complete Futhark program for estimating the value of π.

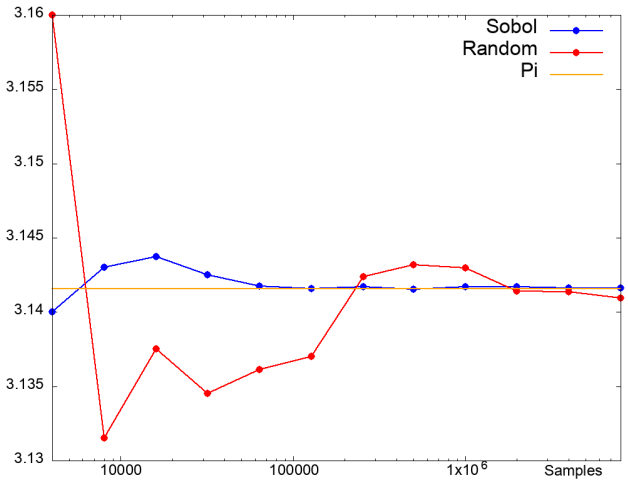

Figure 11. Convergence towards π using Sobol sequences and pseudo-random numbers.

this routine was in OCaml, provided by the financial software company LexiFi[2], and extracted from their commercial offerings. We will refer to the OCaml version as the *reference implementation*. Only a subset, comprising *pure Heston* [23] calibration, has been ported to Futhark.

The calibration routine takes as its main input a series of observed *quotes*. A quote is a triple consisting of a maturity date, a strike price, and a quote price. The model attempts to determine five Heston parameters that match the observed quotes, with the intent that these parameters can then be used to price other quotes in the Heston model.

The fitting of the five Heston parameters is carried out in a conventional manner. Least-squares optimisation is used via the Differential Evolution algorithm [25], where the *objective function* performs closed-form pricing of European options in the Heston model, and compares the projected quotes to the set of observed quotes taken from market data.

Intuitively, the algorithm is an evolutionary (or "genetic") algorithm that proceeds by maintaining a *population* of candidate parameters, which are repeatedly randomly perturbed.

[2] https://www.lexifi.com/

Table 2. Speedup of **`stream_map`** versus fully sequential and fully parallel implementations for computing Sobol numbers. For comparing sequential performance, a compiler generating single-threaded CPU code has been used and the code runs on an Intel Xeon E6-2570. For comparing parallel performance, OpenCL code is executed on an NVIDIA Tesla K40 GPU. We generate 30, 000, 000 Sobol numbers.

Version	Runtime	Speedup
Chunked (executed on GPU)	3.9ms	×11.13
Fully parallel	43.4ms	
Chunked (executed on CPU)	129.7ms	×1.0
Fully sequential	129.1ms	

The random modifications that result in the most improvement (judged via the objective function) are then chosen for further evolution. It is thus hard for a re-implementation to match the reference implementation *exactly*, as it is sensitive to how the randomness is generated. As a result, there is no *single* correct result, as the only measure of algorithmic success is how well the final parameters minimise the error in the objective function. However, as long as a re-implementation is able to generate results of approximately the same quality as the reference solution, it can be considered correct.

The Futhark implementation comprises three main parts: (i) a general-purpose least-squares implementation using the Differential Evolution algorithm, (ii) a European call-option pricer, and (iii) the integration between the first two (plus a slight amount of preprocessing). The vast amount of the run-time work takes place inside the least-squares solver, which uses the European call-option pricer as the objective function. Thus, this is the part on which we will focus.

The least-squares implementation is in principle a reusable component with no specific relation to option pricing. As a result, we have written it to be *generic* in its choice of objective function, even though we only ever apply to the European call-option pricer for this application. We shall see that it is necessary, for some workloads, to exploit both the parallelism provided by the least-squares-process, as well as the objective function, despite these existing at different levels of abstractions.

The most important part of the program contains an outermost **`while`**-loop that encloses several nested parallel loops enclosed within each other, as shown on Figure 12. This figure shows for each level whether the parallel operation is a **`map`** or a **`reduce`**, as well as the symbolic number of iterations performed by that parallel level (see below). The figure is a simplification that focuses only on the most essential structure. For example, it hides various preprocessing operations that have negligible runtime cost. The program is shown in a fused form—the source implementation contains many more **`map`** and **`reduce`** operations that are fused to form the structure shown. Further, the loops are not perfectly nested; some sequential work is for example done for each of the outer `np` iterations of the mutation function before the inner loop is executed.

The parameters controlling the number of iterations of the parallel loops are as follows:

`num_free_vars`: The number of parameters that we are fitting. For the Heston model, this is always 5.

`np`: The population size used for the Differential Evolution algorithm. This is set to 40 in the reference implementation, which we maintain in the Futhark implementation.

`num_points`: The number of Gauss-Laguerre coefficients used for pricing options in the objective function. This is either 20 or 10, but always set to 20 in the reference implementation, which we follow here.

`num_prices`: The number of observed prices. In practice, this is the quantity that can vary between data sets, and thus the only scalable source of parallelism. However, while it is easy to synthesise a large set of prices, is is unclear how large this size is in realistic workloads.

The parallel loops are repeatedly executed by the sequential outer `while`-loop until a *convergence criterion* is reached. The convergence criterion is configurable, and can be either reaching a specified error tolerance, a given number of iterations, or a number of calls to the objective function. For the Heston model, the error tolerance is set to zero and the maximum number of iterations to $2^{31} - 1$. The maximum number of calls to the objective function is set to 2000. In practice, this is the limit that we will reach. The convergence loop invokes the objective function exactly `np` times for each iteration, so it will run for $2000/\texttt{np} = 2000/40 = 50$ iterations.

Many parallel libraries and languages simply do not support nested parallelism at all, typically executing only the outer- or innermost loop in parallel. One reason for this is that supporting arbitrary nested parallelism efficiently is a difficult and generally unsolved problem. However, this particular algorithm exhibits *regular* nested parallelism, which requires that the size of an inner-parallel dimension is invariant to all the outer parallel dimensions.. Regular nested parallelism is much easier to compile efficiently, and is handled by the Futhark compiler through an algorithm called *moderate flattening* [19].

The restriction to regular nested parallelism allows flattening to stop after exploiting only some of the outer-levels of parallelism, which in turn may allow further optimization of locality or efficient sequentialization of operations such as reduce and scans. What is to be sequentialized and what is to be parallelized is currently decided by a crude compile-time heuristic, e.g., inner reduces are parallelized, but inner map-reduce compositions are sequentialized, because they often enable tiling.

In languages that do not support nested parallelism natively, we generally have two options open to us. First, we can *manually* apply some form of the flattening algorithm to produce a flat parallel program. In this case, this would involve violating the abstraction boundary between the least-squares component and the objective function, as we would have to inline the objective function into the least-squares implementation, and then flatten the resulting code. This makes the resulting code non-reusable, which is clearly not desirable.

Another common approach is to simply disregard some of the parallelism. For example, the population size `np` is usually not very large, and so we can sequentialize the outer loop in the recombination function, and exploit only on the parallelism inside the objective function. This keeps the least-squares implementation reusable and generic, but depends on the objective function to supply enough parallelism (in our case, with `num_prices`) to saturate the hardware. As we shall see in section 4.1, this does not perform well for all workloads, and may not perform well with *any* workloads for other applications where the objective function contains little parallelism, and we instead depend on `np` being large.

However, this program does contain a parallel loop that should probably *not* be executed in parallel. Specifically, the **`reduce`** operation in the objective function. If we turn this into a sequential loop, the compiler can generate a simple GPU kernel with no communication between individual threads (corresponding to a nesting of **`map`**s), rather than a more overhead-intensive segmented reduction. As we shall see in section 4.1, this is an improvement because the innermost `num_points` = 20 amount of parallelism is not necessary. While it is unfortunate that the Futhark compiler cannot (yet) do this automatically, manually sequentialising this loop does at least not cross an abstraction layer.

4.1 Performance

We compare the runtime of the Futhark implementation to the reference implementation on three data sets: one with 1062 quotes, one with 10,000 quotes, and one with 100,000 quotes. The latter two are synthetic, but the first data set is representative of the scale of the typical inputs used in practice. Further, we also test a variant of the Futhark implementation where the outer loop of the recombination function has been sequentialised. This is representative of an implementation that does not exploit nested parallelism across abstraction boundaries, as discussed in the previous section. We execute the Futhark implementation on an NVIDIA K40 GPU. The resulting runtimes are shown on Table 3.

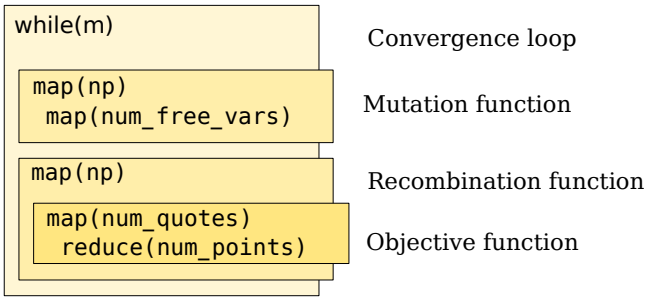

Figure 12. Loop Structure of the Heston Calibration benchmark. The coloured boxes and their labels indicate the logical nesting of the code, and in parentheses the number of iterations. All loops are parallel, except for the outer convergence loop. One loop is a **`reduce`**, while the others are **`map`**s.

Runtime measurements for parallel execution do not include OpenCL/GPU driver initialisation or kernel compilation, nor does it include copying initial input to the device. The time taken for this copying is negligible (less than 1%) compared to the overall runtime.

Speedup for a given dataset and platform is computed by comparing the achieved runtime to the best runtime of the reference implementation on the same dataset. This is the number that indicates how much faster the Futhark implementation is than reference implementation.

It is unsurprising that GPU speedup is generally low on the smallest dataset (1062 quotes), as this dataset contains insufficient parallelism to amortise the overhead of parallel execution on a GPU. If we exploit only inner parallelism, which cuts exploited parallelism in the recombination function by a factor of 40, the speedup falls to an anemic 8.84× on this dataset. However, on the largest dataset, this version is slightly faster (233.65× compared to 198.92× speedup), because the objective function itself has enough parallelism to saturate the GPU. Executing the **`reduce`** in parallel does not affect the smallest dataset, but negatively impacts performance on especially the largest dataset. Despite the relatively efficient implementation of regular segmented reduction used by Futhark [21], we still end up paying an overhead for parallelism that we do not need. The analysis in the remaining section uses the version with a sequentialised **`reduce`**.

5 Related Work

Parallel functional languages, and how to compile them, has been an active field of research for many years. This paper takes a slightly different perspective than most, aimed at demonstrating the benefits of using a high-level data-parallel language for (i) exploring the performance tradeoffs of an application, for (ii) quickly prototyping high-level solutions

Table 3. Runtimes of reference and Futhark implementation on various platforms. Speedups are relative to the fastest reference runtime for a given dataset.

Dataset	Platform	Runtime	Speedup
num_quotes = 1062	Core i5–5300 (OCaml; sequential)	4.60*s*	1.00×
num_quotes = 10,000	Core i5–5300 (OCaml; sequential)	38.54*s*	1.00×
num_quotes = 100,000	Core i5–5300 (OCaml; sequential)	441.60*s*	1.00×
num_quotes = 1062	NVIDIA Tesla K40 (Futhark; only inner parallelism)	0.52*s*	8.84×
	NVIDIA Tesla K40 (Futhark; sequential reduction)	0.047*s*	97.87×
	NVIDIA Tesla K40 (Futhark; all parallelism)	0.059*s*	77.97×
num_quotes = 10,000	NVIDIA Tesla K40 (Futhark; only inner parallelism)	0.64*s*	60.22×
	NVIDIA Tesla K40 (Futhark; sequential reduction)	0.21*s*	183.52×
	NVIDIA Tesla K40 (Futhark; all parallelism)	0.32*s*	120.44×
num_quotes = 100,000	NVIDIA Tesla K40 (Futhark; only inner parallelism)	1.87*s*	236.15×
	NVIDIA Tesla K40 (Futhark; sequential reduction)	2.22*s*	198.92×
	NVIDIA Tesla K40 (Futhark; all parallelism)	3.24	136.3×

to these, and for (iii) building powerful parallel libraries. For example, we have started from the Rodinia implementation of BFS—which n.b., is not expressible in Futhark—and have prototyped other four implementations, which incrementally address observed inefficiencies on specific datasets.

Similarly, we have designed a high-level library for parallel computation of Sobol sequences in Futhark. Previous work has shown how to efficiently combine the independent and recurrent formulas, but they rely on low-level GPU implementations and/or require that Sobol's direction vectors are part of the dataset [1, 22, 24]. In comparison, our library requires the user to only specify the dimensionality of the desired Sobol sequence, and relies on compile-time instantiation of higher-order modules [11] to automate the whole parallel-code generation (direction vectors included).

A significant obstacle limiting the expressivity of many data-parallel languages is the lack of support for nested parallelism. Much effort has been carried out in this area, starting with the seminal work on flattening of nested parallelism in NESL [4, 5] and in more recent work aimed at adapting the transformation for GPU hardware [27] or for multi-core hardware [2], for example by flattening only the data and exploiting dynamic parallelism. Such approaches typically regard all parallelism as highly irregular and are aimed at maximizing the amount of parallelism (e.g., they interchange sequential recurrences outside of the parallel code). However, in doing so they also prevent opportunities for optimising locality or communication (e.g., by tiling). As we saw with Heston (section 4), maximizing parallelism is not the optimal strategy for all data sets.

Other approaches (such as SaC [13, 14], Lime [10], and Accelerate [8]) perform well for flat parallelism, but generally do not support nested parallelism. A problem such as Heston would have to be written in a manually flattened form, which results in non-modular code.

6 Conclusions

We have shown by three examples the advantages of using a high-level hardware independent language for GPU computation. In particular, such a language is (i) well suited to organizing generic libraries and answering modularity concerns, and (ii) allows data-parallel reasoning about the implementation performance and allowing algorithmic tuning of the data-parallel implementation.

We have shown that modular code requires the compiler to exploit multiple levels of parallelism, across abstraction boundaries, to provide the large number of parallel threads required by modern GPUs. For Heston calibration on the real-world dataset (the smallest one), there is an order-of-magnitude difference between exploiting all levels of parallelism, or only the innermost level.

We have demonstrated that beyond a certain (hardware and problem-specific) threshold, further parallelism may induce only overhead, without improving hardware utilization. Thus, it is important that the programming model permits the programmer to specify both fully parallel and more work-efficient sequential algorithm implementations. Here, computation of Sobol numbers shows an order-of-magnitude difference between a maximally parallel and an efficiently sequentialised implementation.

Acknowledgments

We are grateful to LexiFi for providing the Heston calibration benchmark and representative input data, and to NVIDIA for donating the K40 GPU used for this work.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center *HIPERFIT: Functional High Performance Computing for Financial Information Technology* [3] under contract number 10–092299.

[3] http://hiperfit.dk

References

[1] 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.

[2] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the GPU. *SIGPLAN Not.* 47, 9 (Sept. 2012), 247–258. https://doi.org/10.1145/2398856.2364563

[3] Robert Bernecky and Sven-Bodo Scholz. 2015. Abstract Expressionism for Parallel Performance. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2015)*. ACM, New York, NY, USA, 54–59. https://doi.org/10.1145/2774959.2774962

[4] Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge.

[5] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM (CACM)* 39, 3 (1996), 85–97.

[6] 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 (1994), 4–14.

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

[8] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proc. of the sixth workshop on Declarative aspects of multicore programming*. ACM, 3–14.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2254064.2254066

[11] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proc. ACM Program. Lang.* 2, ICFP, Article 97 (July 2018), 30 pages. https://doi.org/10.1145/3236792

[12] Paul Glasserman. 2004. *Monte Carlo Methods in Financial Engineering*. Springer, New York.

[13] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC: A Functional Array Language for Efficient Multithreaded Execution. *Int. Journal of Parallel Programming* 34, 4 (2006), 383–427.

[14] Clemens Grelck and Fangyong Tang. 2014. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*. 129–145.

[15] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language*. Ph.D. Dissertation. University of Copenhagen, Universitetsparken 5, 2100 Kobenhavn.

[16] Troels Henriksen, Martin Elsman, and Cosmin E. 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, USA, 31–42. https://doi.org/10.1145/2636228.2636238

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

[18] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 88, 7 pages. https://doi.org/10.1145/2627373.2627388

[19] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[20] Stephen Joe and Frances Y. Kuo. 2003. Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator. *ACM Trans. Math. Softw.* 29, 1 (March 2003), 49–57. https://doi.org/10.1145/641876.641879

[21] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

[22] 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. *J. Comp. Graph. Stat* 19, 4 (2010), 769–789.

[23] S Mikhailov and U Nögel. 2003. Heston's stochastic volatility model-implementation, calibration and some extensions. *Wilmott magazine* (January 2003), 74–79.

[24] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364474.2364484

[25] 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 (1997), 341–359. https://doi.org/10.1023/A:1008202821328

[26] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

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