Incremental Flattening for Nested Data Parallelism

Troels Henriksen  
University of Copenhagen  
athas@sigkill.dk

Martin Elsman  
University of Copenhagen  
mael@di.ku.dk

Frederik Thorøe  
University of Copenhagen  
vbj532@alumni.ku.dk

Cosmin Oancea  
University of Copenhagen  
cosmin.oancea@diku.dk

Abstract
Compilation techniques for nested-parallel applications that can adapt to hardware and dataset characteristics are vital for unlocking the power of modern hardware. This paper proposes such a technique, which builds on flattening and is applied in the context of a functional data-parallel language. Our solution uses the degree of utilized parallelism as the driver for generating a multitude of code versions, which together cover all possible mappings of the application's regular nested parallelism to the levels of parallelism supported by the hardware. These code versions are then combined into one program by guarding them with predicates, whose threshold values are automatically tuned to hardware and dataset characteristics. Our unsupervised method—of statically clustering datasets to code versions—is different from autotuning work that typically searches for the combination of code transformations producing a single version, best suited for a specific dataset or on average for all datasets.

We demonstrate—by fully integrating our technique in the repertoire of a compiler for the Futhark programming language—significant performance gains on two GPUs for three real-world applications, from the financial domain, and for six Rodinia benchmarks.

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

Keywords  functional language, parallel, compilers, GPGPU.

1 Introduction
Computer performance increases are typically accomplished by increasing capacity for parallelism, which leads to heterogeneous designs that support multiple levels of parallelism and a memory hierarchy that removes the illusion of uniform random-access memory. GPUs are a popular example of this trend. However, programming in low-level GPU-specific languages, such as OpenCL or CUDA, is a daunting task that requires specialized compiler and hardware knowledge and often results in non-modular code that aims at utilising the hardware’s multi-level parallel units and associated memory hierarchy, such as a GPU’s grid of multi-processors, each consisting of a number of parallel computing cores.

A vast amount of work aims at compiling high-level and hardware-agnostic code into efficient low-level code for parallel architectures, thereby shifting the burden from the programmer to a combination of language, compiler, and autotuning infrastructure. For example, various domain-specific languages have been proposed to accelerate the execution of image-processing pipelines [44], iterative stencils [55], data analytics [57], deep-learning [7, 19] and mesh computations [38, 54], graph processing [32, 43], and accelerated host-language constructs [13, 17, 33, 51], but they tend not to support nested parallelism.

Code transformations for improving the amount of nested parallelism that is mapped to hardware have been devised in both imperative and functional contexts, but they cover efficiently only one class of workloads from many possible. For example, the tree-of-bands construction [12, 58] prioritizes locality optimization over utilizing nested parallelism, while full flattening [11] takes the reverse direction. A revised version of full flattening, named “moderate flattening” [31], uses a statically-chosen heuristic to utilize only some of the top-most levels of application parallelism—thus preserving opportunities for further locality optimizations. These approaches still do not judiciously exploit nested parallelism because they depend crucially on the static heuristic to choose good parallelization boundaries, which are often dataset sensitive. Similarly, autotuning strategies compute (i) the optimal combination of compiler flags that, on average, yields the best performance across various datasets [16, 23], or (ii) the best schedule of re-write rules that produces an optimal program for one specific (class of) dataset [44, 50].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPPoPP ’19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6225-2/19/02...
https://doi.org/10.1145/3293883.3295707
short, all related work follows a “one-size fits all” approach that produces one program, potentially parameterized over tile sizes (or similar measures), but which is unlikely to cover well all workloads.

Instead, this paper proposes a compiler-driven analysis that systematically clusters the datasets by a manageable number of piecewise near-optimal code versions, which are discriminated at runtime by statically generated predicates, whose threshold values are autotuned to hardware characteristics. Our principal assumption is that a suitable driver for such clustering is related to the degree of nested parallelism that is utilized and to the way in which it is mapped to the levels of parallelism supported by the hardware.

The main contribution of this paper is a generic flattening technique, organized as a top-down compilation pass over a simple data-parallel language that supports a regular notion of nested parallelism. The pass introduces different guarded code versions at each point where it encounters a map operator that is to be mapped at level \( l \) of hardware parallelism:

1. map the discovered application parallelism onto hardware level \( l \) and sequentialize the inner parallelism,
2. map the discovered application parallelism on hardware level \( l \) and recursively map the remaining inner parallelism at hardware level \( l - 1 \), and
3. recursively continue flattening at hardware level \( l \).

This procedure generates semantically equivalent code versions for all possible utilizations of top-level application parallelism and for all possible mappings to different levels of hardware parallelism. The first two choices are guarded by predicates that compare the amount of parallelism utilized by the corresponding code version with a constant derived by offline training. While the number of generated code versions is exponential in the depth of the parallel nest, we argue that this code expansion is manageable in practice in most cases, as demonstrated by our results. In particular, the language design guarantees that the depth of parallel nesting is determined statically (which is not true for languages that permit recursion), and thus the amount of code explosion is easily estimated by a human programmer inspecting the shape of the program, rather than depending on more subtle quantities.

The proposed compiler-driven autotuning design allows the combination of various code versions into one program, whose behavior can adapt not only to the particularities of different classes of datasets, but also to the dynamic behavior of the sizes of intermediate arrays (levels of parallelism) during one execution of the program. In comparison, related approaches suffer a disconnect between the compiler and the autotuning process, are unable to relate a program to a class of datasets favored by the applied optimisations, and, further on, are unable to combine effectively differently optimised code versions into one program. We claim the following principal contributions:

- A generic flattening algorithm called incremental flattening that combines semantically equivalent code versions covering (i) all possible utilizations of the application’s top-level (nested) parallelism, and (ii) all possible mappings to the hardware’s levels of parallelism (Section 3).
- A specialized training technique that enables quick and efficient clustering of datasets to code versions. Both the incremental flattening techniques and the training/autotuning techniques are fully integrated in the repertoire of a freely available compiler (Section 4).
- A detailed empirical evaluation on two GPU platforms of three real-world financial applications and six Rodinia benchmarks, demonstrating significant performance gains at the expense of a manageable (as high as four times) code-size expansion (Section 5).

Incremental flattening is presented on a simple language supporting regularly nested parallelism by means of the well-known Bird-Meertens operators (map, reduce, scan, etc) [10]. Most data-parallel languages and libraries build on the same formalism. Thus, while our empirical evaluation has been done by extending the compiler for a concrete programming language, Futhark, we believe that our proposed technique is applicable to a wider range of languages.

2 Preliminaries and Motivating Example

The (simplified) source language of the transformation, whose syntax is shown in Figure 1, is a purely-functional first-order expression language, equipped with a sequential semantics, but augmented with second-order parallel array combinators (SOACs), such as map, reduce, and scan (prefix sum).

We assume a denumerable infinite set of variables, ranged over by \( x, y, \) and \( z \), and we use \( d \) to range over integers, \( p \) over variables or integers, and \( b \) over boolean values (true and false). We shall write \( q \) to denote a sequence of objects of some kind, and we shall let the context in which the notation is used govern whether the sequence separator is a comma (,) or simply white space. Binary operators (bop) include the traditional operations on integers and general 0-order operators (op) include a number of operations on multi-dimensional arrays. The expression \( \text{replicate} \ p \ x \) creates an array by replicating the element \( x \) \( p \) times, and \( \text{rearrange} \ (d_1, \ldots, d_n) \ x \) is a generalization of transpose in that it rearranges the dimensions of the array argument, based on a statically known permutation defined by the integer sequence \( d_1, \ldots, d_n \) (e.g., \( \text{transpose} \equiv \text{rearrange} \ (1, 0) \)).

Functions that are passed as arguments to SOACs can be expressed using \( \lambda \)-notation or by partially applying operators (or other SOACs). The loop construct has the semantics of a call to a tail-recursive function; it executes a fixed number of iterations (known before the loop is executed) such that the
Incremental Flattening for Nested Data Parallelism

Figure 1. The syntax of the source language.

Loop parameters \( x_1 \cdots x_n \) are initialized with an expression for the first iteration, and are bound to the result of the previous iteration for the remaining iterations.

We assume A-normal form: let bindings can be seen as a block of statements followed by a sequence of result variables, and the loop initializers are variables or constants; sometimes we diverge from this notation for readability.

For completeness, we present below the type and semantics of well-known SOACs, where we use \([a_1, \ldots, a_n]\) to denote an array literal, and \([n]\) to denote the type of an array of length \( n \) with elements of type \( \alpha \).

\[
\begin{align*}
\text{map} : (\alpha \to \beta) &\to \Pi n. [n]\alpha \to [n]\beta \\
\text{map} f [a_1, \ldots, a_n] &= [f a_1, \ldots, f a_n] \\
\text{reduce} : (\alpha \to \alpha \to \alpha) &\to \Pi n. [n]\alpha \to \alpha \\
\text{reduce} \odot 0_0[a_1, \ldots, a_n] &= 0_0 \oplus a_1 \oplus \ldots \oplus a_n \\
\text{scan} : (\alpha \to \alpha \to \alpha) &\to \alpha \\
\text{scan} \odot 0_0[a_1, \ldots, a_n] &= [0_0 \oplus a_1, \ldots, 0_0 \oplus \ldots \oplus a_n]
\end{align*}
\]

However, the language assumes a tuple-of-arrays representation, which allows SOACs to receive and produce an arbitrary number of parameters and results, as illustrated in the example below, where the map construct operates over and produces two arrays, which are then reduced:

\[
\begin{align*}
\text{let } z_{s_1} z_{s_2} = \text{map} (\lambda x y \to (2 * x, 3 + y)) x x s y s \\
\text{in reduce} (\lambda x_1 x_2 y_1 y_2 \to (x_1 + y_1, x_2 * y_2)) 0 \ 1 z_{s_1} z_{s_2}
\end{align*}
\]

Finally, we define redomap/scanomap as the composition of a reduce/scan with a map according to the equations:

\[
\begin{align*}
\text{redomap} \odot f \overline{d} \overline{s} &\equiv \text{reduce} \odot \overline{d} (\text{map} f \overline{s}) \\
\text{scanomap} \odot f \overline{d} \overline{s} &\equiv \text{scan} \odot \overline{d} (\text{map} f \overline{s})
\end{align*}
\]

2.1 The Target Language

The target language is similar to the source language, but with the key difference that SOACs (e.g., map, reduce, ...) are now understood to execute sequentially. Instead, three new constructs are introduced (segmap, segred, and segscan), which express (and guarantee) parallel execution (modulo the number of processing units available). These new constructs are annotated with a so-called execution level, a static natural number, ranged over by \( l \), which indicates at what level, of the underlying parallel architecture, the construct is executed. Target expressions (\( e \)) and mapnest contexts (\( \Sigma \)) take the following forms:

\[
e : = \ldots \\
\begin{align*}
\text{segmap}^l \Sigma e &\equiv \text{map} (\lambda x_p \to (\lambda x_{p-1} \to \ldots \text{map} (\lambda x_1 \to e) y_1 \ldots) y_{p-1}) y_p \\
\text{segred}^l \Sigma \odot \overline{d} e &\equiv \text{map} (\lambda x_p \to (\lambda x_{p-1} \to \ldots \text{redomap} (\lambda x_1 \to e) \overline{d} y_1 \ldots) y_{p-1}) y_p \\
\text{segscan}^l \Sigma \odot \overline{d} e &\equiv \text{map} (\lambda x_p \to (\lambda x_{p-1} \to \ldots \text{scanomap} (\lambda x_1 \to e) \overline{d} y_1 \ldots) y_{p-1}) y_p
\end{align*}
\]

In essence, the context attached to the parallel constructs records for each nest level the formal parameters of the lambda function and the corresponding arrays from which they take values, while the expression \( e \) is the body of the innermost mapped function. For example, assuming a two-dimensional array (matrix) \( yss = [[1, 2], [3, 4]] \), then \( \text{segmap}^1 \langle x \in yss \rangle (x \in xs) (x + 1) \) adds one to all elements of \( yss \) resulting in array \( [[[2, 3], [4, 5]]] \). Similarly, \( \text{segscan}^1 \langle x \in yss \rangle (x \in xs) (+0 x) \), computes the prefix sums of each row of \( yss \), resulting in \( [[[1, 3], [3, 7]]] \).

The main implicit constraint of the target language is that a parallel construct at level \( l \geq 0 \) can contain only sequential code in its associated "body" expression \( e \), and a parallel construct at level \( l \geq 1 \) can directly contain only parallel constructs at level \( l - 1 \) and sequential code (but cannot directly contain parallel constructs at level \( l \) or \( l - 2 \)).

2.2 Motivation: Matrix Multiplication

Multiplying matrices \( xss \) and \( yss \) can be written as:

\[
\text{map} (\lambda x \to \text{map} (\lambda y \to \text{redomap} (+) 0 x y)) (\text{transpose} yss) xss
\]

Depending on the sizes of the matrices, there are at least three code versions that may result in best performance:

1. If the size of the two outer-levels of parallelism is too small to utilize well the hardware, then the innermost redomap should also be executed in parallel. For example, in the target language, the fully flattened version at hardware level 1 is expressed as:

\[
\text{segmap}^1 \langle x \in xss \rangle \langle y \in \text{transpose} yss \rangle (\langle x y \in xss \rangle (+0 (x \times y))
\]

2. Otherwise, only the two outer map operations should be parallelized and the redomap should be sequentialized.
This would result in the target-language code:
\[
\text{segmap} \,(xs \times xs) \,(ys \in \text{transpose} \, yss) \\
(\text{redomap} \,(+) \,(\text{replicate} \,0 \,xs \,ys)
\]
which enables block tiling for optimizing locality.

(3) If the outer two maps provide parallelism in excess of what the hardware can utilize, then another chunk of parallelism can be sequentialized to further optimize locality by combining register and block tiling.

Figure 2 shows the runtime results on an NVIDIA K40 GPU, for multiplying two matrices of sizes \(2^n \times 2^m\) and \(2^n \times 2^m\), respectively, where \(n = 0 \ldots 10\), \(m = k - 2 \cdot n\), and \(k = 20\) or \(k = 25\); this setup ensures constant workload \(2^k\) in all cases.

The green lines show the result of moderate flattening—an instance of “one size fits all” compiler analysis, which uses version (2) plus block tiling (register tiling is not supported). The black line shows the untuned results of the incremental flattening technique, and the red line shows the results after tuning, for which the datasets for \(k = 20\) were used as training set, and the computed threshold parameters were applied to the \(k = 25\) datasets. Notice that the tuned program accurately gets the best of the two worlds, and it uses code version (1) for \(n < 5\) and code version (2) for \(n \geq 5\). (The compilation used default tile/group-size values; detailed results are available in supplemental material, also for AMD.\(^1\))

Finally, the gray line shows the results of the cuBLAS library, which likely uses multiple versions of (hand-optimized) code, special hardware instructions, a richer optimization repertoire (e.g., register tiling), and superior tuning (tile and block sizes). Still, on \(k = 20\) datasets, incremental flattening is competitive until \(n = 8\). On \(k = 25\) datasets, cuBLAS wins at a large margin for \(n = 4 \ldots 6\), and is \(2 - 3\times\) faster on \(n = 7 \ldots 10\), likely due to register tiling.

This paper does not claim contributions to the compilation of matrix multiplication, but uses it as motivation: Figure 2 shows that a widely used, highly-optimized, and architecture-specific implementation (cuBLAS) of the most used algorithm has suboptimal performance on a class of (degenerate) datasets \((n < 3)\). It is not uncommon for even highly optimised standard implementations of primitives to perform suboptimally on degenerate or exotic input [49]. In practice, even expert programmers lacks either the expertise or, more likely, the manpower necessary to cover comprehensively all datasets of interest with differently-optimized code versions. In more complex cases, such considerations also cross abstraction boundaries (we show an example in Section 5.2), and addressing them by manual code transformation can thus break modularity. This problem opens the door to language and compiler technology to play a prominent role in such a setting; further evidence is presented in Section 5.

### 3 Incremental Flattening Formalization

This section presents a number of rules that transform a nested-parallel program (in the source language of Section 2) into multi-versioned code (in the target language of Section 2.1) that dynamically picks an appropriate version based on the setting of tuning parameters. Section 3.1 discusses the rules of moderate flattening, which were taken from the literature [31] and are not a contribution of this paper. Section 3.2 presents the incremental flattening transformation, which is organized as a generic extension built on top of moderate flattening, and is the core contribution of this paper.

The core inference rules of incremental (and partially for moderate) flattening are formalized in Figure 3. The rules allow inferences of the form \(\Sigma \vdash e \Rightarrow e'\), which are read “in a mapnest context \(\Sigma\), the source expression \(e\) can be translated, at hardware level \(l\), into the target expression \(e'\).” In an inference rule, the part below the line specifies the translation (the conclusion), and the part above the line contains the premises necessary for the translation to fire.

#### 3.1 Moderate Flattening Inference Rules

Rules G4-G7 in Figure 3 belong to moderate flattening, which always executes at hardware level \(l\) (because no rule changes \(l\)). Rule G4 interchanges an outer \(\text{reduce}\) inside an inner \(\text{map}\), which corresponds to the intuition that summing up the columns of an \(n \times m\) matrix can be achieved by transposing the matrix and summing up the elements on each row:

\[
\begin{align*}
\text{reduce} \,(\text{map} \,(+) \,(\text{replicate} \,m \,0)) \,(\text{matrix}) & \equiv \\
\text{map} \,(\text{reduce} \,(+) \,(0)) \,(\text{transpose} \,(\text{matrix}))
\end{align*}
\]

Rule G5 generalizes the observation that transposing each element of an input array can be rewritten as a permutation of the dimensions of the input array:

\[
\text{map} \,(\text{transpose}) \,(\text{arr3d}) & \equiv \text{rarrange} \,[0,2,1] \,(\text{arr3d})
\]

Rule G6 refers to distributing a map nest across the two subexpressions of a \text{let} expression (i.e., map fission). The intuition is given by the equivalence below, which is extended in Rule G6 to operate on entire map nests:

\[
\begin{align*}
\text{map} \,(\lambda \,x \rightarrow \,\text{let} \,y = \,e_1 \,\text{in} \,e_2) \,\text{xs} & \equiv \\
\text{let} \,ys = \text{map} \,(\lambda \,x \rightarrow \,e_1) \,\text{xs} \,\text{in} \,\text{map} \,(\lambda \,x \,y \rightarrow \,e_2) \,\text{xs} \,\text{ys}
\end{align*}
\]

In terms of dependence analysis, this rule corresponds to the observation that it is always safe to distribute a parallel loop across its instructions. In practice, it is not profitable to completely distribute the map nest, so the rule assumes that \text{let} floating and tupling will group scalar expressions together and will partition the \text{let} expression in contexts where a recurrence occurs (e.g., contexts including a construct such as \text{reduce}, \text{transpose}, \text{or loop}) to enable further exploitation of inner parallelism.

Finally, Rule G7 interchanges a \text{map} nest inside a \text{loop}. In terms of dependence analysis, the key observation is that it
is always safe to interchange a parallel loop inwards. The rule generalizes the easy-to-see equivalence:

\[
\text{map } (\lambda x \rightarrow \text{loop } x = x_0 \text{ for } i < n \text{ do } (f x)) x s_0 = \text{loop } x = x s_0 \text{ for } i < n \text{ do } (\text{map } f x)
\]

The rule covers all possible invariant \((q, z)\) and variant \((\overline{x}, \overline{y})\) variables to the outer \text{map}, and requires that the loop count is invariant. The intent of this rule is to enhance the degree of utilized parallelism by merging the parallelism that exists outside and inside the \text{loop}; as such the rule is applied only when the body of the loop contains parallel constructs.

These rules, in addition to changing the schedule of instructions, also performs array expansion (G6 and G7), and provide hints for changing the array layout (G4 and G5).

A significant shortcoming of moderate flattening is that it is formulated in terms of a nondeterministic choice of where actually to stop flattening, that is, where to manifest the mapnest context \(\Sigma\) over \(e\). This problem does not show in Figure 3 as Rules G1-G3 belong to incremental flattening.

In practice, this nondeterminism suggests that the algorithm needs to be augmented with a static (compiler) heuristic that fixes the behavior; for example, an inner \text{redomap} will be sequentialized, while a \text{reduce} perfectly nested in a \text{map} nest will be parallelized, and so on. The ability not to fully apply flattening is essential for performance, as it permits further optimisation of locality (e.g., by block tiling) and thread communication (e.g., by sequentializing the SOACs of logarithmic depth such as \text{reduce, scan}). However, the static heuristic can be highly inaccurate in many cases of practical interest—because the parallel sizes are typically statically unknown (dataset specific)—and as such it may lead to severe underutilization of hardware parallelism.

### 3.2 Incremental Flattening Inference Rules

This section presents the core contribution of this paper, a (generic) extension applied on top of moderate flattening, which solves the nondeterministic specification of the latter in a simple way, by generating all possible mappings of (top-level) application parallelism to all possible hardware levels. The resulting semantically-equivalent code versions are combined into a single program by guarding them with predicates that compare the amount of parallelism utilized by each with a threshold value, which is later autotuned.

Figure 3 presents the rules of the incremental flattening transformation. Rules G0, G1, and G2 are the base cases of a recursive algorithm. Rule G0 implements the identity in the case when the mapnest context is empty and no other rule can be applied—we recall that the parallel constructs of the source language have sequential execution in the target language. Rule G1 says that if we are already at level 0 under a non-empty context and no other rule applies then we manifest the map nest on top of the current expression.

Rule G2 treats the case in which the mapped expression does not contain any inner parallelism. Since there are no opportunities for further flattening, the existing parallelism is manifested by means of a \text{segmap} construct.

Rule G3 is the core of the transformation; whenever a \text{map} that contains nested recurrences is discovered, three code versions are produced. One, named \(e_{\text{top}}\), manifests the context plus the current map at the current hardware level and sequentializes the body of the current map. Another one, named \(e_{\text{middle}}\), manifests the map nest and the context at the current level of hardware, but it recursively attempts to utilize the inner parallelism of the current map at a decremented hardware level. Finally, the last code version, namely \(e_{\text{flat}}\) is obtained by continuing flattening at the current hardware level. In the figure, \(\text{Par}(\Sigma)\) and \(\text{Par}(e)\) correspond to expressions that symbolically represent the (maximal) degree of parallelism available in a context and in a target expression, respectively, and \(t_{\text{top}}\) and \(t_{\text{intra}}\) are (free) variables that can be considered program arguments, and their near-optimal values are found by autotuning. The remaining rules (G4-G7) are the same as in the moderate flattening case, which demonstrates the generic nature of our extension.

Furthermore, Figure 4 shows two additional rules that, we speculate, were not included in the repertoire of moderate flattening because they were not profitable in the so called “common” case, but we found practical uses for them under the more protective umbrella of incremental flattening.

Rule G8 exploits inner parallelism inside a branch, by interchanging the context’s map nest inside the \text{then} and \text{else}
Uses hardware parallelism levels: \( l, l - 1, \ldots, 0 \). 

\[ \Sigma \vdash l \iff e' \]

\[ \Sigma \vdash e \Rightarrow e \]  

\[ \Sigma \vdash \text{segmap}^0 \Sigma e \]  

\[ \Sigma \vdash \text{map} (\lambda \bar{x} \to e) \Rightarrow e \]  

\[ \Sigma \vdash \text{segmap}^l \Sigma' e' \]  

\[ \Sigma \vdash \text{reduce} (\text{map} (\lambda y \to e)) (\text{replicate } k d) \bar{x} \Rightarrow e' \]  

\[ \Sigma \vdash \text{rearrange} (0, 1 + k, \ldots, 1 + k_n) y \Rightarrow e \]  

\[ \Sigma' \vdash \text{let } \bar{a} = e_0 \iff e_1 \text{ in } e_2 \Rightarrow \bar{a} = e'_1 \text{ in } e'_2 \]  

\[ \Sigma' \vdash \text{loop } \bar{z}' y' \Rightarrow e \text{ for } i < n \text{ do } f i \bar{q} \bar{x} y' z' \Rightarrow e \]  

Figure 4. More aggressive incremental-flattening rules.

Figure 3. Rules for the incremental flattening extension.

More Incremental Flattening Rules

- The innermost \( \text{map} \) from the context \((\bar{x} \in \overline{y})\) and deriving the mapped expression \((\Sigma' \vdash l \Rightarrow e')\) so that Rule G3 will immediately try all three choices. Rule G6 can be made more aggressive with a similar refinement.

We recall that moderate flattening would sequentialize any inner \( \text{redomap} \) (e.g., to enable block tiling). Rule G9 shows the new treatment that (i) generates a code version \( e_{\top} \), which manifests existential parallelism by a \( \text{segred} \) construct, and then it (ii) decomposes the \( \text{redomap} \) into a \( \text{map} \) and a \( \text{reduce} \) that are recursively processed to exploit inner levels of parallelism. (A not-shown rule is that a \( \text{redomap} \) exhibiting no inner parallelism is directly manifested by a \( \text{segred} \)).

Finally, we remark that while our implementation of the presented rules targets GPU hardware (see next section), we believe they at least set a solid foundation for approaching other types of heterogeneous hardware, such as multicore with SIMD support. Notice also that, even for GPU hardware, the presented approach is far from optimally solving the “one size fits all” problem, for example because our solution considers only one important driver for multi-versioned code generation (the degree of utilized parallelism) from many (e.g., locality, thread divergence, load balancing). However, our approach still covers indirectly some cases of locality optimization, because for example block tiling in scratchpad memory typically sacrifices (sequentializes) lower level(s) of parallelism, and the combination with register tiling sacrifices yet another chunk of outer parallelism, which is sequentialized and moved into the innermost context.

Typing rules for the source and target languages and a type-preservation theorem for incremental flattening are available in the supplemental material.

4 Implementation and Autotuning

Full Futhark supports arrays of tuples, polymorphism, higher-order functions [34], and higher-order modules [21], but a series of conventional defunctionalisation and monomorphisation transformations produces first-order programs of a form
very similar to the one presented in Section 2. In particular, all functions are inlined, arrays-of-tuples are transformed to tuples-of-arrays, and aggressive fusion \([29, 30]\) is performed prior to flattening \([31]\). Futhark also supports more SOACs (e.g., \texttt{filter} and \texttt{scatter}), array operations (e.g., reshaping and slicing), and in-place updates, but these operations are omitted here as they do not interact essentially with flattening. The modularity and abstraction features of Futhark allow for programmers to reason at a high level about the performance and the composition of a program \([28]\), while, at the same time, Futhark can be used as a high-performance target language for domain-specific languages \([3, 26]\).

### 4.1 GPU Code Generation

In a nutshell, a GPU program is organized into a grid of equally-sized \textit{workgroups}, each of which executes an equal number of parallel threads. The number of logical threads spawned is thus the number of workgroups times the workgroup size. We model the GPU as having two levels of parallelism: one at grid level \((l = 1)\) and one at workgroup level \((l = 0)\). This distinction does not mean that a program containing only level-1 parallel constructs uses less parallelism than a semantically-equivalent one that contains a nest of level-1 and level-0 constructs. For example, consider the following fully-flattened expression at level 1:

\[
\text{loop } xs = xs_0 \text{ for } i < n \\
\text{segmap}_i (xs \in xs_0) (x \in xs) (x + 1)
\]

This expression utilizes all parallelism, but stores the input array \(xs\) and the result array in global memory, which makes it significantly slower than the equivalent expression:

\[
\text{segmap}_i (xs_0 \in xs_0) (\text{loop } xs = xs_0 \text{ for } i < n) \\
\text{segmap}_i (x \in xs) (x + 1)
\]

What is exploited at level 0 is in fact locality of reference, because the threads in a workgroup can be synchronized (by barriers) and they can collectively use fast local memory, rather than global memory, which is at least one order of magnitude slower. All intermediate arrays produced by parallel operations at hardware level 0 are allocated in fast local memory. Unfortunately, local memory capacity is sharply limited, typically on the order of 32 to 64KiB depending on the GPU. Execution fails if a workgroup requires more local memory than available. Our implementation does not currently address this issue, but there are two obvious ways to handle it. One option is to add a clause to Rule G3 such that we only select \(e_{middle}\) if its local memory requirements are less than what is supported by the hardware. A second option is to dynamically fall back to storing intermediate arrays in global memory. This can be implemented by generating a “fallback kernel” that substitutes global memory for local memory in the final OpenCL code. Thus it has the advantage that memory pressure does not directly impact which parallelization structure we end up selecting, but it is not clear whether there is a good reason to parallelise at level 1 if intermediate results have to be stored in global memory.

### 4.2 Autotuning

Code generated with incremental flattening picks between different code versions based on symbolic \textit{threshold parameters}. These must be assigned a concrete value when the program is run. By default, all thresholds are assigned the value 215, as a rough estimate of how much parallelism is needed to saturate a GPU. However, this is likely suboptimal.

Because an exhaustive test of all possible parameter settings is not viable, we have implemented a simple stochastic autotuner via the OpenTuner framework \([4]\). We define one tunable \texttt{LogIntegerParameter} for every threshold parameter. This parameter class presents a log scaled view of the parameter to the search technique, such that halving and doubling the parameter appears as changes of equal magnitude. For some setting of threshold parameter values, we then run the program on a series of user-provided datasets, and apply a \textit{cost function} to the observed program runtimes. Our goal is to find the parameter settings that minimise the cost function. For this paper, our cost function simply sums the runtimes for all datasets, which favours improvements on large datasets. However, our autotuner does not depend on any specific cost function, and a different measure could easily be employed, or even provided by the user. In practice, a weighted sum would be a good choice, as it permits the user to indicate which workloads are the most important.

Unfortunately, the search space for an incrementally flattened program is highly repetitive: Different parameter settings may result in the same dynamic behavior for a dataset. For example, consider the branching tree on Figure 5. It shows a hypothetical Futhark program that has been flattened to four different versions, each guarded by a predicate comparing a threshold parameter \(p_i\) to some dataset-dependent size \(n_i\). Suppose that for some dataset, \((n_1, n_2, n_3) = (10, 20, 30)\). The parameter assignment \((p_1, p_2, p_3) = (5, 15, 25)\), results in version V1, but so does assignments with \(p_1 = 6\)!

It would be a waste if the autotuner reran the program just to repeat a path that has been tried before. Unfortunately, there is no way to express such constraints directly with

![Sample branching tree produced by incremental flattening. For space reasons, the size of this this tree has been pruned, and it does not correspond directly to any program shown in the paper.](image)
OpenTuner. Instead, we pass OpenTuner a cost function that knows the structure of the branching tree for the program (which is provided by the Futhark compiler), and if OpenTuner attempts a parameter assignment that would result in a path through the tree that has been attempted before, the function returns the previously observed runtime immediately, which allows OpenTuner to resolve such duplicate parameter assignments very quickly.

While our autotuner provides good results quickly for many programs, it is ultimately still stochastic, and for large and complicated programs we have observed that it may still take a very long time for it to discover the optimal parameter settings—or possibly time out before finding them at all. This is because the attempted parameter settings are essentially random. An improvement would be to use the structure of the branching tree to avoid redundant parameter settings entirely, but such irregular search spaces are difficult to encode in OpenTuner, and a fully exhaustive search might still be too slow on large programs.

5 Experimental Validation

The experimental validation is divided into two parts. In the first part (Section 5.2) we analyze in detail the LocVolCalib benchmark (from the FinPar suite [2]), which provides good insights into how incremental flattening works.

In the second part (Section 5.3), we perform bulk validation on eight benchmarks, two computational kernels used in real-world pricing of financial options (code provided by the company LexiFi) and other six benchmarks from Rodinia. The benchmarks were chosen to highlight the benefits of incremental over moderate flattening: on other benchmarks (not shown) their performance is very similar. This section uses the following abbreviations: MF and IF for moderate and incremental flattening and AIF for autotuned IF.

5.1 Experimental Setup

We run all benchmarks on two different Linux systems, one with an NVIDIA K40 GPU, on CUDA 8.0, and one with an AMD Vega 64 GPU, on AMDGPU-PRO 18.20. We perform auto-tuning separately on the two systems. As we shall see, parameters that are optimal for one, are not necessarily optimal for the other. The K40 supports OpenCL group sizes up to 1024, while Vega 64 supports group sizes up to 256. We let the autotuner run for 20 minutes per benchmark, albeit most of them require less than 1 minute to find optimal thresholds.

The datasets used for tuning are different than the ones used for testing; their choice was based on application specific knowledge. To simplify the tuning phase, we perform no tuning of tile and workgroup sizes, which use default values (256 threads per group). Based on manual experimentation, tuning these sizes results in performance improvements of at most around 10%. On average, IF takes 4× longer to compile and generates 3× larger binaries than MF.

We measure total application runtime, minus the time taken for (i) loading program input onto the GPU, (ii) reading final results back from the GPU, and (iii) OpenCL context creation and kernel build time. Excluding these fixed overheads emphasizes the performance differences between implementations. Any other host-device or device-device communication/copying/memory allocation is measured.

Runtimes were averaged across 10 runs, the maximal observed standard deviation was 3%, and the results of Futhark’s moderate flattening as presented in [31] are used as baseline.

5.2 The LocVolCalib Benchmark

LocVolCalib is derived from real-world stochastic volatility calibration [2]. The code structure, shown in Figure 6a, consists of an outer map of degree numS, containing a sequential loop of numT iterations, the body of which contains two maps of the function tridag over the arrays xss (size [numY][numX]) and yss (size [numY][numX]). The tridag function is a composition of three scans (see Figure 6b). Together, numT, numS, numX, and numY characterize the workload, with the latter three contributing to parallel work. The three datasets are:

- small (numS = 16, numT = 256, numX = 32, numY = 256),
- medium (numS = 128, numT = 64, numX = 256, numY = 32),
- large (numS = 256, numT = 64, numX = 256, numY = 256).

On most hardware we have tried, for small and medium, the parallelism inside tridag must be exploited, while for large, tridag may be sequentialised.

The structure of the generated code and the three versions that get exercised by datasets are shown in Figure 6c. They were obtained as follows: First, the third choice in Rule 3 makes the outermost map part of the context and applies flattening recursively (the other two choices are not interesting for the current datasets). Second, Rule G7 interchanges the map context inside the loop. Third, Rule G4 performs the distribution of the outer map (part of the context) across the two inner maps. At this point, a new map is encountered, that is, the one producing xss’. The first, second, and third choices of Rule G3 produce the three code versions: (1) on the branch numS*numX > t1 it parallelizes the two outer maps at hardware level 1 and it sequentializes tridag, (2) on the branch numS*numX*numY > t2 it also parallelizes the three scans inside tridag in scratchpad memory at workgroup level 0, and (3) on the else branch, it utilizes all parallelism at hardware level 1 under the form of three segmented scans. In principle version (2) requires two global-memory accesses per data element for executing all three scans, and is thus more efficient than version (3) that requires at least two (and typically three) global-memory accesses per data element for each of the three scans.

Results are shown in Figure 7. The last two bars correspond to two (efficient) hand-written OpenCL implementations [2]: the first, named FinPar-Out, is similar to version 1, and the second, named FinPar-All, is similar to version 2.
Incremental Flattening for Nested Data Parallelism

```plaintext
map (λxss0 yss0 →
    loop xss, yss = xss0, yss0
    for t < numT do
        let xss' = map tridag xss
        let yss' = map tridag yss
        in (xss', yss'))
(xss0: [numS][numX][numY][f32])
(yss0: [numS][numY][numX][f32])
```

(a) Simplified structure of LocVolCalib.

```
tridag as = let bs = scan @ d_0 as
    let cs = scan @ d_0 bs
    in scan ⊙ d_0 cs
(b) Parallel formulation of tridag.
```

if numS > t0 then ... else

```plaintext
loop xssss, yssss = xssss0, yssss0
for t < numT do
    let xssss' =
    if numS*numX > t1 then
        -- Version 1 utilizes numS*numX
        -- parallelism at hwd level l=1
        segmap1 (xss ∈ xss)(xss ∈ xss)(tridag xs)
    else if numS*numX parallelism at l=1
        -- Version 2:
        -- numS*numX parallelism at l=1
        -- and numY parallelism at l=0
        segmap1 (xss ∈ xss)(xss ∈ xss)
            (let bs = segscan^0 (x ∈ xs) ⊙ d_0 x
             let cs = segscan^0 (b ∈ bs) ⊙ d_0 b
             in segscan^0 (c ∈ cs) ⊙ d_0 c)
    else -- Version 3: numS*numX*numY > t2 then
        -- parallelism at hwd level l=1
        let bsss=segscan^1 (xs ∈ xss)(xs ∈ xss)
            (x ∈ xs) ⊙ d_0 (x)
        let csss=segscan^1 (bs ∈ bs)(bs ∈ bs)
            (b ∈ bs) ⊙ d_0 (b)
        in segscan^1 (css ∈ css)(css ∈ css)
            (c ∈ cs) ⊙ d_0 (c)
    let yssss' =... --code similar to xssss'
in (xssss', yssss')
```

(c) The three generated versions that get executed in practice.

Figure 6. Parallel structure of the LocVolCalib program, and how incremental flattening exploits its parallelism.

The large dataset demonstrates the performance portability problem: FinPar-Out wins on K40 but loses on Vega 64; this is because the Vega 64 is in relative terms more memory bound (than K40), favoring local-memory utilization.

MF always generates version 3 of the code (segmented scans executed in global memory). AIF choses version 2 on Vega 64, and a combination of code versions on K40. For both the small and medium datasets, AIF uses Versions 1 and 2 for the tridags of innermost size 32 and 256, respectively, and for the large dataset it uses version 1. Figure 7 shows that (i) AIF significantly outperforms MF in all cases, (ii) on Vega 64, AIF is slightly slower than FinPar-All in all cases, due to suboptimal memory reuse, and (iii) on K40, AIF is faster than FinPar on the small and medium datasets—because it combines versions 1 and 2—but is outperformed by FinPar-Out on the large dataset. The slowdown is mainly due to the fact that FinPar-Out uses an algorithmically-different sequential version of tridag, which performs significantly less work (in number of global-memory accesses). Writing this specialized version in Futhark is possible and gets within 93% of the FinPar-Out performance, but it looses the ability to generate from one specification, multiple parallel versions.

For a software-engineering perspective, FinPar’s implementations (~ 1500 lines each) are very difficult to maintain since they have little resemblance to the algorithmic (nested-parallel) specification, which on the other hand is faithfully reproduced by the Futhark source program (~ 200 lines).

5.3 Bulk Validation

This section shows the utility of incremental flattening on a wider range of benchmarks. Each benchmark uses two datasets (D1/D2 in Table 1) chosen to exhibit different distributions of parallelism. When available, we report speedup of a reference implementation written in OpenCL C. The speedups, using MF as baseline, are shown in Figure 8.
The first two benchmarks are used in real-world financial applications. OptionPricing [2, 39] exhibits several layers of nested parallelism, where D1 is best executed, using only the outermost parallelism, and D2 requires utilization of inner layers as well. The reference implementation utilizes only the outer parallelism, which explains the slowdown on D2.

Heston is a calibration routine for the Hybrid Stochastic Local Volatility / Hull-White model. The original implementation is in sequential OCaml and a hand-written OpenCL implementation is not available. Heston contains three layers of parallelism, an outer map, which contains a reduce operator. MF exploits only the outer map—the static heuristics sequentializes redmaps—which results in poor performance. IF exploits all parallelism, which is optimal on Vega 64, but is suboptimal on K40. AIF gives the best results, by sequentializing and parallelizing the innermost reduce on K40 and Vega 64, respectively.

The other six benchmarks are from Rodinia [15], of which Backprop, LavaMD, and NW already contain nested parallelism. In Backprop, for MF, we have explicitly prevented a fusion between an inner map and reduce, which otherwise would have resulted in poor performance (redmaps are sequentialized). AIF wins because of fusion; Rodinia’s slowdown is due to a reduce being executed on the CPU.

In LavaMD, both Rodinia and MF exploit only two outer levels of parallelism, and they tile in local memory an inner redomap contained in a sequential loop, which is optimal on D1. On D2, AIF wins because it also parallelizes the inner redomap (at workgroup level) in local memory.

In NW, Rodinia’s implementation processes in parallel waves of blocks on lines parallel to the matrix’s diagonal, where a block is parallelized at workgroup level in local memory. AIF does the same; the ∼ 2x slowdown is due to the matrix update not executing in place (diagonal slices are not expressible in Futhark). MF has poor performance because it flattens aggressively without utilizing local memory.

The Futhark ports of the remaining three Rodinia benchmarks (NN, SRAD, Pathfinder) have been extended with an extra layer of parallelism by adding a map on top; essentially performing multiple batches of the original benchmark in parallel. We have not modified the original Rodinia programs, so we can only report Rodinia performance for the D1 datasets, for which the factor of outer parallelism added is 1. Speedup on the D2 datasets is due to exploiting more parallelism, especially at intra-workgroup level. Rodinia’s poor performance on NN and Pathfinder are due to an important reduce being executed on CPU (NN), and due to pyramidal tiling (Pathfinder) that does not seem to pay off on the tested hardware. The supplemental material gives more insight related to the parallel structure of the benchmarks.

It would have been interesting to compare against a compiler making use of full flattening, but unfortunately no such compiler targeting GPUs is currently available. As an approximation, we modified the heuristics used by MF to always fully exploit parallelism. For these benchmarks, the resulting programs are typically slower within a factor 2 of untuned incremental flattening, but for e.g. OptionPricing the runtime is more than an order of magnitude higher, because a large amount of redundant nested parallelism is being exploited.

6 Related Work

Full flattening of irregular nested parallelism was introduced in the seminal work on NESL [11]—also implemented on GPU hardware [9]—but performed poorly in practice due to high memory usage and communication costs caused by excessive parallelization. Later efforts have focused on addressing these issues by restricting flattening in various ways, for example by flattening only the data and leaving the nested-parallel structure intact [8], by applying flattening at
the granularity of the largest sequential subexpression [35],
by aggressive fusion of segmented operations enabled by
shape analysis [27, 47], or by mechanisms for streaming ir-
regular arrays [18, 36] that optimize memory footprint. In
comparison, incremental flattening is the first to enable re-
stricted utilization of parallelization in the common case,
while still permitting full parallelization if necessary.

Polyhedral compilation [42] has been proven effective in
optimizing affine programs. For example, the tree-of-bands
algorithm [12, 58] optimizes both the degree of parallelism
and locality of references, but locality takes precedence when
they conflict. Furthermore, algorithmic lower bounds for
locality can be modeled analytically at the level of misses in a
set associative cache hierarchy [6]. Since the affine domain is
too restrictive in practice, several techniques were devised to
integrate explicit annotations of parallelism with polyhedral
transformations of otherwise unanalyzable patterns [5, 14,
46].

Autotuning has been motivated by the performance-
portability problem in the context of changing hardware.
One class of solutions [23] has been to apply machine learn-
ing, relying on supervised off-line training, to infer the best
configuration of compiler flags that results in the average-
best performance across various datasets and hardware; en-
couraging results have been reported for multi-core [16]
and many-core architectures [5], which, in some cases, are
shown, for example, to outperforming cuBLAS.

Other approaches promote a compiler design reliant on
autotuning. For example, Halide [44] applies a stochastic
method to find the best fusion schedule of image-processing
pipelines, corresponding to various combinations of tiling,
sliding window and work replication transformations. Simi-
larly, Lift [24, 50] and SPIRAL [22], exploits the rich rewrite
rules of functional languages to generate a multitude of
equivalent programs, from which autotuning selects the op-
timal one for a given dataset.

In comparison, this paper proposes an unsupervised method
in which the incremental application of flattening, statically
clusters the datasets to corresponding code versions based on
the amount of utilized parallelism. This permits recombining
all versions into one program that covers all datasets.

Optimising task-based nested parallelism requires tech-
niques that control the granularity at which parallelism is
exploited in order to avoid unnecessary overhead. Notably,
Heartbeat scheduling [1] uses a provably efficient sched-
uling algorithm and does not require tuning. Perhaps the
closest related work [56] to ours proposes a combination of
(i) a compiler transformation that generates a set of (multi-
versioned) task implementations of increasing granularity by
means of recursive task unrolling, and (ii) a runtime heuristic
for automatically choosing the best suited implementation.
However, such techniques essentially rely on the hardware
efficiently supporting dynamic exploitation of parallelism,
which makes them unsuitable for GPUs.

Multi-versioned code generation has been used in other
contexts, for example automatic parallelization of non-affine
loops (Fortran77 or C) on multi-cores, where inter-procedural
analysis is used to summarize statically memory references
under a set-abstraction representation and to model loop in-
dependence as an equation on such sets [25, 48]. To improve
accuracy, summaries were paired with predicates [37] that
specify the conditions under which summaries are known to
be empty. A more aggressive variant has been to use logical
inference rules [40, 41] to translate the loop independence
equation to a general predicate encapsulating arbitrary pro-
gram slices. To minimize the runtime overhead, this predicate
was factorized into a set of sufficient conditions that were
tested at runtime in the order of their complexity.

This paper combines static and dynamic analyses in a simi-
lar manner, but here multi-versioning refers to parallel code
optimized differently in the presence of nested parallelism,
while the predicates are trivial and subject to autotuning.

7 Conclusions and Future Work
We have presented incremental flattening, a generic exten-
sion to the flattening algorithm for regular nested data par-
allelism, that is able to turn inner parallelism in excess into
efficient sequential code when profitable, while still having
the ability to exploit all available parallelism when dataset
and execution hardware requires it.

We have shown a refinement of standard autotuning which
is capable of quickly selecting good thresholds for the pred-
icates that discriminate these code versions, and we have
validated our approach on ten problems where performance
is sensitive to the dataset characteristics and hardware.

A natural direction for future work is to support irregular
nested parallelism. In such cases, it is likely that other per-
formance parameters become increasingly important, such
as thread divergence and load imbalance. We speculate that
a solution may draw inspiration from the inspector-executor
techniques used for optimising locality/communication by
reordering both the data and the iteration space [20, 45, 52,
53]. For example inspectors can compute a measure of such
properties, which can be integrated in the predicates that
guard code versioning. The rationale would be that flattening
improves both divergence and load balancing at the expense
of locality, and making them part of the predicates allows this
trade-off to be tuned across various datasets and hardware.

Acknowledgments
We are grateful to NVIDIA for donating the K40 GPU used for
this work, and to P. Sadayappan for providing constructive
criticism on the manuscript. A major source of inspiration
for this work has been Lawrence Rauchwerger’s sensitiv-
ity analysis. This research has been partially supported by
the Independent Research Fund Denmark grant under the
research project FUTHARK: Functional Technology for High-
performance Architectures.
References


Incremental Flattening for Nested Data Parallelism

York, NY, USA.


A Artifact Appendix

A.1 Abstract

This artifact contains scripts and programs for reproducing the central experimental figures of the paper (Figures 2, 7 and 8), which show how Futhark programs perform compared to hand-written GPU programs, using different compilation techniques.

This artifact requires access to a computer with a modern GPU (a laptop will likely not cut it) with OpenCL and optionally CUDA already set up and working. Apart from that, the software requirements are fairly standard. A Dockerfile is provided that will work on Linux systems with NVIDIA GPUs, assuming NVIDIA’s docker runtime has been installed. This is typically not problematic.

The artifact is structured around a Makefile with targets for the figures and supporting data. The enclosed README.md contains further details and hints about execution and troubleshooting. The artifact is also publicly available at https://github.com/diku-dk/futhark-ppopp19.

A.2 Artifact check-list (meta-information)

- **Program**: Rodinia and FinPar; automatically downloaded and patched by scripts.
- **Compilation**: Requires gcc (or compatible) able to compile OpenCL programs; optionally also nvcc.
- **Run-time environment**: Requires Linux (some parts may work on macOS). Docker image may require root access (unclear; we are not Docker experts). A UTF-8 enabled locale is required (e.g. set environment variable LC_ALL=en_US.UTF-8).
- **Hardware**: We require a recent-ish NVIDIA or AMD GPU, ideally with 4GiB or more memory (README describes how to run the less memory-intensive parts on smaller GPUs). Some of our dependencies may require an x86 CPU.
- **Execution**: Full execution takes from one to three hours depending on system speed.
- **Metrics**: We measure wall-clock runtimes, and the enclosed plotting scripts measure speedup compared to a computed baseline.
- **Output**: We produce speedup graphs in PDF format, and raw measurement data in a simple JSON format.
- **Experiments**: A Dockerfile is provided for use on NVIDIA systems, which has also been made available on Docker Hub for ease of use. Ideally, running the experiments and generating the graphs is just make.
- **How much disk space required (approximately)?**: The artifact itself is small, but running it will take more than 10GiB, but less than 20GiB.
- **Code/data licenses (if publicly available)?**: ISC (non-copyleft free license).
- **Archived?**: At ACM Digital Library; DOI: 10.1145/3300173.
A.3 Description

A.3.1 How delivered

The artifact is available (with further documentation and details) from the ACM Digital Library with DOI 10.1145/3300173, and at https://github.com/diku-dk/futhark-ppopp19. It is also available on Docker Hub under the name futhark/ppopp19.

A.3.2 Hardware dependencies

A somewhat modern GPU with at least 4GiB of memory. A significant subset of the experiments can be run with just 2GiB of memory.

A.3.3 Software dependencies

OpenCL to communicate with the GPU, and optionally also CUDA.

A.3.4 Data sets

Included in artifact (or downloaded automatically as needed).

A.4 Installation

Using Docker (recommended for x86-64) On a Linux machine with an NVIDIA GPU, the CUDA framework, and NVIDIA's docker runtime\(^4\), run the following command to enter a Docker instance with the artifact and its various requirements pre-installed:

docker run -it --runtime=nvidia \ 
  --storage-opt size=20G futhark/ppopp19

This may require root permissions, depending on the local system. The artifact also contains a packaged Docker image file, futhark-ppopp19.docker. This image is self-contained, in that it contains pre-downloaded versions of all resources that are otherwise downloaded from the Internet (datasets and benchmark reference implementations), so its contents can be useful for machines that are offline, or if the required resources disappear from the Internet in the future.

Installing Manually Unpack the artifact archive or clone the Git repository, taking care to also include the submodules that contain the Futhark compiler and the FinPar benchmark suite:

git clone --recursive \ 
  https://github.com/diku-dk/futhark-ppopp19.git

To install the software dependencies on a Debian/Ubuntu system, run the following commands (possibly as root):

apt install binutils build-essential nvidia-cuda-toolkit \ 
  sqlite3 librsqlite3-dev libltinfo-dev python-pip git curl \ 
  wget bc libffi-dev libgmp-dev zlib1g-dev texlive \ 
  texlive-latex-extra texlive-fonts-recommended \ 
  dvipng locales

pip install opentuner matplotlib

curl -sSL https://get.haskellstack.org/ | sh

In case of ambiguities, the Dockerfile also serves as a list of commands necessary to install the required packages.

A.5 Evaluation and expected result

- make matmul-runtimes-large.pdf and
  make matmul-runtimes-small.pdf should construct two

\(^4\)https://github.com/NVIDIA/nvidia-docker#quickstart

A.6 Experiment customization

Adding a new Futhark implementation of a benchmark simply requires adding a .fut program to the benchmarks directory, with a specially formatted header indicating which workloads to use. Consult the existing programs or the Futhark User’s Guide\(^5\) for more information. Adding a new dataset requires modifying the header for one of the existing .fut files. By convention, the program benchmarks/foo.fut has its datasets in benchmarks/foo-data. After a program benchmarks/foo.fut is added, it can be benchmarked by running

make results/foo-moderate.json \ 
  results/foo-incremental.json \ 
  results/foo-incremental-tuned.json

Adding a new reference implementation of a benchmark (e.g. one from Rodinia) is done in an ad-hoc fashion. In particular, Rodinia implementations must be manually modified with instrumentation to perform measurements. The main plotting script (tools/bulk-impact-plot.py) is relatively easy to extend with new benchmarks and datasets (albeit with exactly two datasets per benchmark) by modifying the programs variable.

A.7 Notes

While Futhark code tends to be fairly robust in foreign environments, we cannot guarantee the same for the reference benchmark implementations (FinPar and Rodinia) we use. The README.md in the Git repository contains information on how to run only the Futhark implementations, if necessary.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20180713.html
- http://cTuning.org/ae/reviewing-20180713.html
- https://www.acm.org/publications/policies/artifact-review-badging

\(^5\)https://futhark.readthedocs.io