# Mapping Parallelism in a Functional IR through Constraint Satisfaction

## A Case Study on Convolution for Mobile GPUs

Naums Mogers
University of Edinburgh, UK
naums.mogers@ed.ac.uk

Lu Li
University of Edinburgh, UK
lu.li@ed.ac.uk

Valentin Radu
University of Sheffield, UK
v.radu@sheffield.ac.uk

Christophe Dubach
McGill University, Canada
christophe.dubach@mcgill.ca

## Abstract

Graphics Processing Units (GPUs) are notoriously hard to optimize for manually. What is needed are good automatic code generators and optimizers. Accelerate, Futhark and Lift have demonstrated that using a functional approach is well suited to solve this challenge. Lift, for instance, uses a system of rewrite rules with a multi-stage approach. Algorithmic optimizations are first explored, followed by hardware-specific optimizations such as using shared memory and mapping parallelism.

While the algorithmic exploration leads to correct transformed programs by construction, it is not necessarily true for the latter phase. Exploiting shared memory and mapping parallelism while ensuring correct synchronization is a delicate balancing act, and is hard to encode in a rewrite system. Currently, Lift relies on heuristics with ad-hoc mechanisms to check for correctness.

This paper proposes to extract parallelization constraints automatically from a functional IR and use a solver to identify valid rewriting. Using a convolutional neural network on a mobile GPU as a use case, this approach matches the performance of the ARM Compute Library GEMM convolution and the TVM-generated kernel consuming between 2× and 3.6× less memory. Furthermore, a speedup of 12× is achieved over the ARM Compute Library direct convolution implementation.

*CC '22, April 02–03, 2022, Seoul, South Korea*

ACM ISBN 978-1-4503-9183-2/22/04. . . $15.00
https://doi.org/10.1145/3497776.3517777

***CCS Concepts:*** • **Computer systems organization** → *Neural networks*; • **Software and its engineering** → **Source code generation**; *Parallel programming languages*.

***Keywords:*** code generation, convolution, mobile GPU, parallelism

**ACM Reference Format:**
Naums Mogers, Lu Li, Valentin Radu, and Christophe Dubach. 2022. Mapping Parallelism in a Functional IR through Constraint Satisfaction: A Case Study on Convolution for Mobile GPUs. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3497776.3517777

## 1 Introduction

Parallel architectures such as Graphics Processing Units (GPUs) are notoriously hard to optimize for. Programs have to be written in low-level languages such as OpenCL, which expose architectural details including shared memory, threads, synchronization primitives and vectorization. As a result, many automatic approaches have been proposed for code generation and optimizations.

However, producing high-performance parallel code automatically is challenging. There are many optimizations that need to be considered (*e.g.*, tiling, coalescing, prefetching), and many ways to map data (*e.g.*, shared memory) and computation (*e.g.*, work groups, threads), leading to a large implementation space. Different approaches have been proposed to address this problem. TVM [4] relies on the user to specify the desired schedule. Polyhedral compilers [2, 22, 24] automate exposing parallelism, but rely on heuristics and internal performance models to choose a parallel mapping. Accelerate [13] and Futhark [8], two functional approaches, rely on hard-coded heuristics to choose a parallelization strategy.

Lift uses a different mechanism where optimization choices are expressed via rewrite rules and a search of the space is performed via sampling [20]. To tackle the large search space, Lift relies on a multi-stage approach

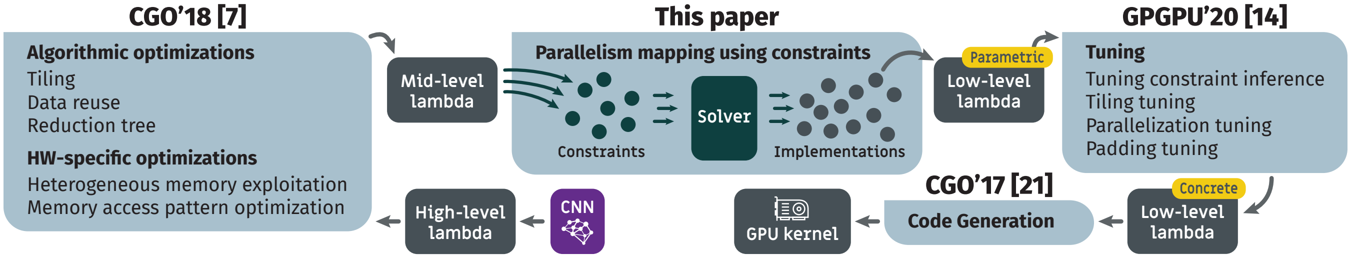


**Figure 1.** The entire optimization flow in Lift.

where algorithmic optimizations such as tiling are first explored using rewriting. This is followed by hardware-specific optimizations such as using shared memory and mapping parallelism using the same rewriting system.

While the algorithmic exploration phase results in transformed programs that are correct by construction, extra effort is required for the latter phase. Parallelization involves side-effects that are hard to account for with the fine-grained rewrite rules of Lift. Most Lift papers shy away from this problem and, like many, the current Lift compiler relies on hard-coded heuristics combined with ad-hoc mechanisms to guide this process. Although this practical approach produces high-performance code, it is far from being an ideal state of affairs.

This paper proposes a new approach to mapping parallelism in the context of the data-parallel functional Lift IR. It reformulates the problem as a constraint satisfaction problem encoding most of the restrictions of the programming model. Crucially, rewrite rules are still used to perform the exploration of the space, but the rewrites producing invalid mappings of parallelism can be avoided. By automatically generating parallelization constraints, the compiler prunes away invalid implementations from the search space.

To evaluate this new approach, the VGG-16 [19] Convolutional Neural Network (CNN) is used as a use-case on a mobile GPU. The focus is on the convolution – the most compute-intensive operation [10] of a CNN architecture. Prior work [14] has shown how this kernel can be expressed and optimized in Lift. In contrast to prior work, the mapping of parallelism is performed automatically using constraints. Convolution implementations require high levels of loop nesting especially after tiling and thus present an additional challenge to parallelize.

The experimental results collected on ARM Mali GPU show that this new approach outperforms the handwritten ARM Compute Library [12] direct convolution kernels by 12× and is on par with its GEMM method while using 3.6× less memory. It also achieves performance on par with the state-of-the-art TVM [4] code generator while using 2.7× less memory, which is important in the context of mobile GPUs.

The main contributions of this paper are:

- Parallelization constraint generation capturing scheduling restrictions specific to individual loops;
- Automatic parallelization of VGG-16 on ARM Mali GPU, achieving performance on par with TVM and memory savings of more than 2×.

The rest of this paper is organized as follows. The next section gives an overview and motivation while Section 3 introduces background information about convolution and Lift. Section 4 presents the core contribution of mapping parallelism in a functional IR with constraints while Section 5 evaluates our new approach. Section 6 discusses related work and Section 7 concludes.

## 2 Overview and Motivation

Figure 1 presents the entire Lift optimization flow. Starting from a high-level expression, the input program is first transformed at the algorithm level by applying optimizations such as tiling. Then, hardware-specific optimizations come into play such as optimizing memory access patterns or exploiting shared memory and mid-level lambda is produced. The resulting mid-level lambda expresses several structural optimizations and presents multiple opportunities for parallelization, which are exploited in the second stage.

In contrast to prior Lift work [7], this paper separates parallelism mapping into its own stage, which produces a low-level parametric parallelized lambda. In the final stage, the lambda is auto-tuned, using the same approach presented in the most recent Lift paper [14]. This results in a low-level expression, which is vectorized and passed to the Lift OpenCL code generator [21].

### 2.1 Challenge of Mapping Parallelism

As we will see shortly, Lift exposes parallelism opportunity through the use of the `map` IR primitive which corresponds to a loop. During rewriting, the `map` primitive can be replaced by a parallel loop implementation, exploiting different levels of parallelism in the architecture (*e.g.*, work groups, local threads, vectorization) or turned into a sequential implementation. Herein lies the challenge of the search space explosion: treating each loop

```
1  for i in 1 to I do                               // Loop A
2    for j in 1 to J do                             // Loop B
3      for k in 1 to K do                           // Loop C
4        local_buf[j][k] = f( input[i][j][k] );
5    for k in 1 to K do                             // Loop D
6      for j in 1 to J do                           // Loop E
7        output[i][j][k] = g( local_buf[j][k] );
```

**Listing 1.** Example parallelizable expression

as an independent parallelization opportunity results in a large number of invalid mappings of parallelism.

Consider the code in Listing 1 that we wish to map onto the OpenCL parallelism hierarchy. In OpenCL, parallelism exists at the global level (G0,G1), or in a combination of work group (W0,W1) and local levels (L0,L1). In Listing 1, `f` is applied on a global input where the results are stored in a local buffer, which is then read to produce a global result by applying `g`. Listing 1 contains five loops (A-E) that can be mapped in numerous ways. We now look at six (non-exhaustive) possible mappings listed in Table 1.

**Naive approach M1** parallelizes the outer loop across global threads. Although valid, this might lead to a lot of sequential work or perhaps little parallelism depending on the loop iterations count. Since the iteration number could be a tuning parameter (*e.g.*, dependent on tile size) this could be an interesting design point nonetheless, as certain tuning values could lead to good performance.

**Mapping M2** parallelizes loops A, B and D across all global OpenCL threads in two dimensions. However, because of a data dependency, a global barrier is required between lines 4 and 5. Since OpenCL does not support global synchronization, this mapping is invalid.

**Mapping M3** produces out-of-scope reads since local data cannot be shared between work groups.

**Mapping M4** schedules work groups across the outer loop A, and the compiler can insert a local synchronization barrier between loops B and D. However, the barrier could impose a performance penalty.

**Mapping M5** vectorizes loop E. However, since vectorized loads require contiguous data in memory and the threads are accessing non-contiguous elements in line 7, such vectorization is invalid.

**Mapping M6** is valid and ensures that each thread accesses only the results it produced itself, thus eliminating the inter-thread data dependency and the need for the barrier. Vectorization is applied on the contiguous access in line 4. This mapping might lead to good performance.

This example demonstrates that naive parallelization strategies can produce invalid code. Manual scheduling of kernels with hundreds of loops is costly and poorly generalizable. Furthermore, although invalid kernels could be detected during code generation, early detection of invalid parallel mappings is desired to avoid the overhead of tuning invalid kernels. This paper tackles the problem by modeling parallelization restrictions in Lift using constraints and finding valid mappings using a solver.

**Table 1.** Example parallel mappings for Listing 1. **Gn**, **Wn** and **Ln** stand for global, work group and local parallelizations respectively in the OpenCL dimension `n`. S stands for *sequential* and **V** – for *vectorized.*

| | Loop parallelization | | | | | Parallel mapping assessment |
|---|---|---|---|---|---|---|
| | A | B | C | D | E | |
| M1) | **G0** | S | S | S | S | Under-saturated cores |
| M2) | **G1** | **G0** | S | **G0** | S | Invalid: cannot synchronize |
| M3) | S | **W0** | **L0** | **W0** | **L0** | Invalid: out-of-scope reads |
| M4) | **W0** | **L0** | S | **L0** | S | Synchronization overhead |
| M5) | **W0** | **L0** | S | **L0** | **V** | Invalid: unvectorizable |
| M6) | **W0** | **L0** | **V** | S | **L0** | Might be optimal |

## 3 Convolution in Lift

CNNs depend on the convolution operation to produce feature maps, *i.e.*, tensors characterizing spatial distribution of visual features in the input image; the maps are used in subsequent layers for classification. In CNN architectures such as SENet [9], up to 99.99% of inference run time is spent computing convolutional layers, which is why this paper focuses on optimizing convolution.

In a convolutional layer, features are encoded using trained weight tensors called *convolutional kernels.* A feature map is obtained by sliding a kernel across both spatial dimensions of the image. Each slided window is convolved with the kernel weights across all input channels. An output value is produced by summing all weighted values of the slided window.

The two widely adopted convolution algorithms are the *General Matrix Multiply* (GEMM) method and *direct convolution.* The GEMM method uses the *im2col* operation to produce a copy of each sliding window stacked with other windows as columns in a single matrix. This allows performing convolution by multiplying the reshaped input and weights in a single GEMM operation, for which many Basic Linear Algebra Subprograms (BLAS) libraries provide optimized kernels. However, GEMM increases memory consumption due to data duplication. In VGG layer 2, im2col enlarges the input from 13 MB to 116 MB. This puts a significant strain on resource-constrained platforms such as mobile GPUs.

The direct convolution method is based on a stencil algorithm, which updates elements based on their neighboring values. Although the direct approach uses less memory, the input access patterns are more complicated than in GEMM, requiring careful optimization of memory access patterns.

```
1  def conv( in: [[[T]_{inChs}]_{inW}]_{inH},
2            ks: [[[[T]_{inChs}]_{kerW}]_{kerH}]_{outChs},
3            kerStepX: int, kerStepY: int
4          ) : [[[T]_{outChs}]_{outW}]_{outH} =
5   mapND_2(slideWin: [[[T]_{inChs}]_{kerW}]_{kerH} ⇒
6     map(singleK: [[[T]_{inChs}]_{kerW}]_{kerH} ⇒
7       reduce(+, 0, map(*,
8         joinND_2(zipND_3(slideWin, singleK)))),
9       ks), slideND_2(kerH, kerStepY, kerW, kerStepX, in))
```

**Listing 2.** Lift expression of direct convolution. Base type $T$ is *float*

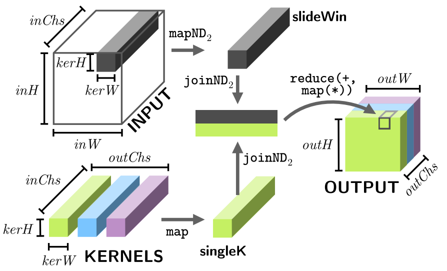


**Figure 2.** Visualization of direct convolution

### 3.1 Functional Expression of Direct Convolution

Listing 2 presents a functional expression of direct convolution in the Lift language; Figure 2 shows a visual representation of the same expression. The expression applies on 3D inputs and 4D kernels. Inputs are defined across multiple channels; kernels are defined across input and output channels, where each output channel represents a feature. $\texttt{slideND}_2$ on line 9 slides a 2D-window across the input, reshaping it from $(inH, inW, inChs)$ to $(outH, outW, kerH, kerW, inChs)$, where $inH$, $inW$ and $inChs$ are input height, width and number of input channels respectively; $outH$ and $outW$ are the output height and width respectively and also the numbers of sliding windows vertically and horizontally. $kerH$ and $kerW$ are kernel height and width respectively. The 2D-`map` on line 5 applies its inner function – a *lambda* – on rows and columns sliding windows, while the `map` on line 6 iterates across output channels. Each combination of a sliding window and a single kernel is reduced to a single value by pairing the corresponding input values and kernel weights using $\texttt{zipND}_3$, flattening the resulting 3D array of tuples, multiplying elements of each tuple and summing weighted values in an accumulator initialized to zero.

$\texttt{mapND}_n$, $\texttt{zipND}_n$, $\texttt{joinND}_n$ and $\texttt{slideND}_n$ are Lift macros, which are automatically expanded into equivalent lower-level Lift expressions and are transparent to the Lift compiler. The `(p: T ⇒ e)` notation denotes a lambda with a parameter of type $T$ and body expression $e$. The next section discusses Lift IR further.

### 3.2 Lift IR

The Lift functional data-parallel language abstracts away the complexities of hardware, shifting the optimization burden from users to the compiler. Expressions such as the one in Listing 2 declare only what needs to be done, ignoring the implementational details. For example, the `map` pattern does not enforce a particular parallelization strategy, and inputs, accumulator and outputs address spaces are also not specified. The functional patterns of Lift create a rich algorithmic representation of the given problem, helping the compiler to perform radical optimizing transformations of a program. Lift IR is discussed in detail in previous work [7, 14, 21].

***Control Structures.*** *map* and *reduce* are the two main control structures in Lift: both are compiled to OpenCL for-loops. *map* applies $f$ on every element of the input while *reduce* "folds" the result using an initialized accumulator and a binary function.

While the *reduce* pattern is always sequential, Lift complements the sequential `mapSeq` with several parallel variants for OpenCL: `mapGlobal`, `mapWrg` and `mapLcl`. These variants capture the OpenCL programming model, where work can be parallelized across a flat thread index domain (global), work groups and threads within work groups (local). For each of the three domains, OpenCL permits parallelizing in three dimensions. For local and global domains, dimension 0 indexes neighboring threads.

***Data Layout Transformers.*** A number of primitives in Lift express data layout transformation patterns that only affect the shape of the data without changing its value. These include *split*, which subdivides an input array into chunks, *join*, which flattens outermost array dimensions, and *transpose*, which swaps two outermost array dimensions. *zip* transforms a tuple of arrays into an array of tuples; for example, a $\texttt{zipND}_2$ has the following signature: $([[T]_b]_a,\ [[U]_b]_a) \Rightarrow [[(T,U)]_b]_a$. Convolution in particular depends on *slide*, which creates an extra array dimension by sliding a window of a given size across the outermost input dimension with a given step [7]. Lift keeps track of such virtual transformations using its *View* system which associates each subexpression with read and write memory mapping functions that can emit index expressions taking into account the history of data layout transformations.

***Address Spaces.*** Lift captures the OpenCL memory model using primitives: `toGlobal`, `toLocal` and `toPrivate`. Wrapping an expression in these forces the output into global, shared or private memory.

### 3.3 Rewriting

The Lift IR yields well to automated program analysis thanks to the lack of side effects and its high-level program representation. The compiler leverages these properties using *rewrite rules* — local semantics-preserving transformations of the AST which are defined on specific patterns. A set of a few dozens of such rewrite rules can be used to create a search space covering design decisions required to achieve high performance.

Since this paper focuses on modeling the parallelization restrictions, navigating the search space of other optimizations is outside the scope of this work. We assume that a set of rewrite rules are chosen heuristically to optimize the input program, short of mapping parallelism which is the focus of this work:

- Tiling input image and weights multiple times allows to exploit *data locality* at multiple levels;
- *Data is reused* by prefetching many input tiles and kernels before entering the loops on lines 5–6;
- *Memory hierarchy* is exploited by storing the results of prefetched data in shared or private memory and by accumulating in shared or private memory when reducing;
- *Memory access patterns* are optimized by inserting transposition before and after prefetching as well as across reduction trees;
- *Weight kernel partitioning* is performed to increase locality and, sometimes, improve access patterns;
- *Vectorization* is applied exhaustively to 1D `maps` with consecutive memory accesses. To establish whether accesses are consecutive, Lift checks the differences between index expressions at iterations $i,\ i+1,\ ..,\ i+(vectorLen-1)$.

Applying the optimizations listed above, the rewriting process creates over 200 parallelizable loops (in the form of *maps*). The next sections discuss how the exposed parallelism is automatically exploited and mapped.

## 4 Parallelization Constraint Generation

State-of-the-art heuristic parallelization methods focus on defining the prospective parallelizing strategies. This approach falls short when presented with new parallel architectures and exotic applications. This section discusses an alternative approach of capturing the restrictions of the target. Invalid parallel mappings can be avoided by automatically generating constraints based on a given AST. Valid parallelizations are free of data races, respect the memory scoping rules and the parallelism hierarchy.

The constraints discussed here encode parallelization restrictions present in many programming models such as OpenCL, CUDA and OpenMP. We use OpenCL as a shared-memory execution model use case, but the methodology is not restricted to this model.

**Table 2.** Encoding of map transformation choices

| Code value | Map transformation |
|---|---|
| 0 | `mapSeq` |
| 1 | Fused with the outer `map` |
| 10, 11, 12 | `mapLcl` in dimension 0, 1 or 2 respectively |
| 20, 21, 22 | `mapWrg` in dimension 0, 1 or 2 respectively |
| 30, 31, 32 | `mapGlb` in dimension 0, 1 or 2 respectively |

The parallelization pass begins by traversing the given expression in search of `maps`, which are used by Lift IR to express parallelization opportunities. Each `map` is associated with a new arithmetic parameter representing a choice of schedule. Then, the search space is restricted with a set of constraints on the new parameters, built using expression types, views, memory allocation, AST structure and target hardware limitations. Any constraint solver library that supports all predicates described in Section 4.2 can be used to generate a restricted search space of implementations to be traversed using established search techniques.

### 4.1 Scheduling Choices

**Sequential loops** are created by replacing `map` with `mapSeq`. They create vectorization opportunities and extend the lifetime of a thread.

**Parallel loops** distribute work across all threads (`mapGlb`), work groups (`mapWrg`) or threads within a work group (`mapLcl`), indexed across three OpenCL dimensions. The dimension choice is explored to achieve memory coalescing since threads within a warp are in dimension 0.

**Map fusion** can be applied on chains of perfectly nested `maps`. Parallelizing fused maps allows distributing more work across threads. Fusion is achieved by replacing `map(map(f))` with `(split(..) o map(f) o join)`.

The `map` transformation parameters are defined on an integer range representing scheduling choices, enabling the use of well-optimized integer constraint solvers. Table 2 provides the parameter encoding scheme. As we will see later, this encoding allows distinguishing between levels of parallelism using division by 10, and between parallelization dimensions – using modulo 10.

### 4.2 Constraint Generation

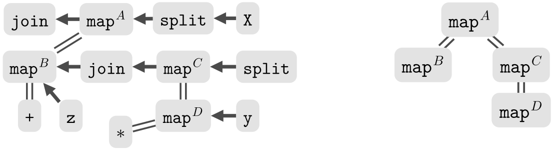
In the context of rewriting Lift programs, a constraint is a predicate restricting the range of values of one or more integer parameters representing design choices:

```
Constraint( parameters: List⟨Parameter⟩,
            predicate: List⟨Int⟩ => Boolean )
```

Where `List⟨T⟩` denotes a list of elements of type `T`, and `=>` denotes a compiler-level function. The emitted predicates are logical conjunctions of quantifier-free

```
join o map^A(map^B(_ + z) o join o map^C(map^D(_ * y)) o
           split(s1)) o split(s0) $ (X: [T]_n)
```

**Listing 3.** An example LIFT expression. `$` denotes application and `o` denotes composition. Letters A-D are unique loop identifiers used to refer to the corresponding `map` instances later on in text. The underscore symbol in `map(_ + z)` denotes the `map` lambda parameter



**(a)** **(b)**

**Figure 3.** The AST (a) and the corresponding `map` nesting tree (b) of the expression in Listing 3; arrows denote composition, double lines denote nesting

equality and inequality constraints over nonlinear integer domain. The supported operators express comparison (>, <, ≥, ≤, =, ≠), integer arithmetic (+, −, /, ×, %) and logical operators: `AND` (∧), `OR` (∨) and `NOT` (¬). Constraints are defined either manually to express OpenCL and hardware limitations and heuristics, or generated automatically to preserve program semantics.

The number of parallelizable `maps` and their positions in the AST are not known in advance since rewriting is performed before scheduling, and rewrite rules can add, relocate and eliminate `maps` in the program candidates. This makes it necessary to collect contextual information of the current expression before constraints can be generated. The contextual functions listed in Table 3 are used to choose the `maps` to generate constraints for.

Consider an example function $f$, which multiplies each element of the array `X` by a scalar value `y` and adds the scalar `z` to each element of the resulting array:

$$f(X : [T]_N, \quad y : T, \quad z : T) = X * y + z$$

Listing 3 is one possible implementation of $f$, in which multiplication is double-tiled and summation is single-tiled. The corresponding `map` nesting tree shown in Figure 3b can be used to provide the first four functions in Table 3. The `MapNestingChain` set, for example, would contain chains (`map`$^A$, `map`$^B$) and (`map`$^A$, `map`$^C$, `map`$^D$). `ConcreteMaps` is collected by checking the nested user functions; memory usage is inferred from the `toLocal` and `toPrivate` primitives. The three last functions return the dimensions of parallelism for each map construct in a nesting chain.

Most contextual functions in Table 3 are known to the compiler during constraint generation from just the given lambda and can be used to decide which constraints to generate. The three last functions, however, are based on the chosen `map` parallelizations, and can therefore be evaluated only in the solving phase, once the solver parallelizes the relevant `maps`. These functions need to be expressed in a way a solver can parse, *i.e.*, as logical conjunctions integrated into the constraints themselves. For example, when some constraint `C` must be enforced for each global dimension used among (`map`$^A$, `map`$^B$), the production rule generating `C` is equivalent to:

**Table 3.** Contextual functions for constraint generation

| Contextual function | Result |
|---|---|
| `NestedMaps(m)` | All `maps` nested in `map m`. |
| `OuterMaps(m)` | All `maps` wrapping `map m`. |
| `MapNestingChains` | All `map` chains from the outer to the innermost nested `maps`. |
| `m1.perfectlyNestedIn(m2)` | True if `map m1` is perfectly nested in `map m2`. |
| `ConcreteMaps` | All `maps` that write to memory. |
| `m.usesPrivateMemory`,<br>`m.usesLocalMemory` | True if `map m` accesses private or local memory respectively. |
| `GlbDimsUsedIn(maps)`,<br>`WrgDimsUsedIn(maps)`,<br>`LclDimsUsedIn(maps)` | Global, work group or local dimensions respectively used in `maps`. |

$$\forall \texttt{g} \in \texttt{GlbDimsUsedIn}(\texttt{map}^A,\ \texttt{map}^B) \ : \ \texttt{C}(\texttt{map}^A,\ \texttt{map}^B) \iff$$
$$\bigwedge_{\texttt{g} \in \{0,1,2\}} \texttt{C}(\texttt{map}^A,\ \texttt{map}^B) \ \vee \ (\texttt{mapEncoding}(\texttt{map}^A) \neq 30 + \texttt{g} \ \wedge \ \texttt{mapEncoding}(\texttt{map}^B) \neq 30 + \texttt{g})$$

Where $\bigwedge$ denotes conjunction, and `mapEncoding(m)` returns the encoding of `m` according to Table 2. This means that for each parallel dimension `g`, either the constraint `C` must hold, or the dimension `g` must not be used in any of the `maps`. Abstracting these extra conjunctions away as stand-alone contextual functions leads to concise production rules.

The contextual information is used to generate six types of constraints that are satisfied only by programs that adhere to OpenCL programming model and are data race-free. In the absence of a formal definition of a correct OpenCL program, the constraints discussed below attempt to capture the restrictions listed in the OpenCL documentation [6].

### 4.3 Memory scoping constraints

Parallel programming models often restrict memory types to specific parallel levels. In OpenCL, private memory is accessible to a single thread, while local memory is shared across threads in a work group, but not across work groups; global memory is accessible on all levels. On a GPU, private, local and global memories can correspond to registers, compute core SRAM blocks and DRAM respectively. Due to differing access speeds and capacities of the three memory types, an optimal memory mapping for a memory-bound application is heterogeneous and specific to the target platform. An automatic parallelization method must produce valid implementations irrespectively of the memory mapping.

***Private memory scoping.*** `maps` that consume or produce private memory cannot be parallelized since private memory is restricted to a single thread. For example in Listing 3, if $\texttt{map}^{D}$ writes the output into registers, both $\texttt{map}^{D}$ and its consumer $\texttt{map}^{B}$ must be executed within the same thread, *i.e.*, only the outer $\texttt{map}^{A}$ can be parallelized. Thus, if parameter `y` of $\texttt{map}^{D}$, or $\texttt{map}^{D}$ output are in private memory, $\texttt{map}^{D}$ cannot be transformed into `mapGlb`, `mapWrg` or `mapLcl`.

A constraint must be generated for $\texttt{map}^{D}$ that allows only the mappings where $\texttt{map}^{D}$ is sequential or fused with an outer `map`. A constraint for such `maps` can be generated as follows:

$$\begin{array}{l}\forall\texttt{m} \in \texttt{ConcreteMaps},\ \texttt{m.usesPrivateMemory}\\ \quad\textbf{GEN CONSTRAINT: } \texttt{mapEncoding(m)}/10 < 1\end{array} \tag{1}$$

This represents a production rule defining which `map` or combination thereof to generate which constraint for. Rule 1 generates the constraint for the `maps` that access private memory, and `mapEncoding(m)/10 < 1` requires that such `maps` are not parallel.

***Shared memory scoping.*** Local memory scoping requires that shared memory is only accessed by threads executed on the same compute core. In Listing 3, the multiplication output could be put in local memory; in that case, a legal parallelization would be following:

$$\begin{array}{l}\texttt{.. o mapWrg}^{A}\texttt{(0)(mapLcl}^{B}\texttt{(0)(toGlobal(\_ + z)) o .. o}\\ \quad\texttt{mapLcl}^{C}\texttt{(0)(mapSeq}^{D}\texttt{(toLocal(\_ * y))) o ..) o .. \$ X}\end{array}$$

The following parallelization would be illegal:

$$\begin{array}{l}\texttt{.. o mapGlb}^{A}\texttt{(1)(mapGlb}^{B}\texttt{(0)(toGlobal(\_ + z)) o .. o}\\ \quad\texttt{mapGlb}^{C}\texttt{(0)(mapSeq}^{D}\texttt{(toLocal(\_ * y))) o ..) o .. \$ X}\end{array}$$

This constraint is expressed as follows:

$$\begin{array}{l}\forall\texttt{m} \in \texttt{ConcreteMaps},\ \texttt{m.usesLocalMemory},\\ \quad\forall\texttt{Chain} \in \texttt{MapNestingChains},\ \texttt{m} \in \texttt{Chain},\\ \quad\quad\forall\texttt{w} \in \texttt{WrgDimsUsedIn(Chain)}\\ \quad\quad\quad\textbf{GEN CONSTRAINT: } (\texttt{mapEncoding(m)}/10 < 2) \land\\ \qquad\qquad\qquad\qquad\displaystyle\bigvee_{\texttt{mOuter} \in (\texttt{Chain} \cap \texttt{OuterMaps(m)})} \texttt{mapEncoding(mOuter)} = 20 + \texttt{w}\end{array} \tag{2}$$

Where $\bigvee$ denotes inclusive disjunction, `m ∈ Chain` restricts the constraint to the chains that include `m` and `mOuter` is one of the outer `maps` of `m`. The `WrgDimsUsedIn` term can only be evaluated by the constraint solver once the search begins, so the constraint is supplemented by predicates enumerating all outer `maps` of `m`.

Rule 2 ensures that `maps` consuming or producing shared memory are local or sequential. It also ensures that such `maps` are uniquely assigned to a single work group by outer instances of `mapWrg`.

### 4.4 Hierarchical Parallelism Constraints

The hierarchies of parallelism in parallel hardware present extra challenge in scheduling computation. The levels of parallelism must be mapped exhaustively, unambiguously, and conforming to the hierarchy. This section focuses on three parallelism levels of OpenCL: global, work group and local.

***Duplicate scheduling constraint.*** `maps` that are directly or indirectly nested cannot be parallelized in the same domain and dimension. In Listing 3, $\texttt{map}^{C}$ and $\texttt{map}^{D}$ cannot be parallelized equally, and the same for other `map` nests. Duplicate scheduling is prevented as follows:

$$\begin{array}{l}\forall\texttt{m} \in \texttt{ConcreteMaps},\ \forall\texttt{mInner} \in \texttt{NestedMaps(m)}\\ \quad\textbf{GEN CONSTRAINT: } (\texttt{mapEncoding(m)}/10 < 1) \lor\\ \qquad\qquad\qquad\qquad\qquad(\texttt{mapEncoding(m)} \neq \texttt{mapEncoding(mInner)})\end{array} \tag{3}$$

The disjunction term `mapEncoding(m)/10 < 1` ensures that the constraint is applied to the parallel `maps`.

***Local thread indexing dimensionality constraint.*** Dimensions used for thread indexing within a work group must match dimensions used for work group indexing and vice versa. For example, if an expression uses `mapLcl(0)` and `mapLcl(1)` and not `mapLcl(2)`, it must also use `mapWrg(0)` and `mapWrg(1)`, but not `mapWrg(2)`. It is evident from the nesting tree in Figure 3b that the maximum depth of `map` nesting in the example expression is three; the only allowed parallelizations are therefore a `mapWrg` nesting a `mapLcl` in the same dimension, or up to three nested instances of `mapGlobal` using different dimensions. This restriction is expressed as follows:

$$\begin{array}{l}\forall\texttt{Chain} \in \texttt{MapNestingChains},\\ \quad\forall\texttt{l} \in \texttt{LclDimsUsedIn(Chain)}\\ \quad\quad\textbf{GEN CONSTRAINT:}\\ \qquad\displaystyle\bigvee_{\texttt{w} \in \texttt{WrgDimsUsedIn(Chain)}} \texttt{w} = \texttt{l}\end{array} \tag{4}$$

$$\begin{array}{l}\forall\texttt{Chain} \in \texttt{MapNestingChains},\\ \quad\forall\texttt{w} \in \texttt{WrgDimsUsedIn(Chain)}\\ \quad\quad\textbf{GEN CONSTRAINT:}\\ \qquad\displaystyle\bigvee_{\texttt{l} \in \texttt{LclDimsUsedIn(Chain)}} \texttt{l} = \texttt{w}\end{array} \tag{5}$$

***Local indexing hierarchy constraint.*** User functions must be parallelized across work groups before they can be parallelized across work group threads. In Lift, this means that no `mapLcl` can have a `mapWrg` with the same dimension nested inside, and each `mapLcl` must be nested in a `mapWrg` with the same dimension.

In the example in Listing 3, this constraint permits $\texttt{map}^{A}$, $\texttt{map}^{B}$ and $\texttt{map}^{C}$ to be made `mapWrg(0)`, `mapLcl(0)` and `mapLcl(0)`, but not `mapLcl(0)`, `mapWrg(0)` and `mapWrg(0)` respectively. This is expressed as follows:

$$\begin{array}{l}\forall\texttt{m} \in \texttt{ConcreteMaps},\ \forall\texttt{mInner} \in \texttt{NestedMaps(m)}\\ \quad\textbf{GEN CONSTRAINT:}\\ \quad\quad\neg((\texttt{mapEncoding(m)}/10 = 1) \land\\ \quad\quad\quad(\texttt{mapEncoding(mInner)}/10 = 2) \land\\ \quad\quad\quad(\texttt{mapEncoding(mInner)}\%10 = \texttt{mapEncoding(m)}\%10))\end{array} \tag{6}$$

***Exhaustive thread indexing dimensionality constraint.*** All user functions must be parallelized in the same number of dimensions to leave no ambiguity in work distribution among threads. For the two nesting chains in Figure 3b, this requires that if $\texttt{map}^A$ and $\texttt{map}^B$ are mapWrg(0) and mapLcl(0) respectively, there must also be a mapLcl(0) among $\texttt{map}^C$ and $\texttt{map}^D$.

$$\begin{aligned}&\forall\texttt{ChainA} \in \texttt{MapNestingChains}, \forall\texttt{ChainB} \in \texttt{MapNestingChains},\\ &\quad\forall\texttt{l} \in \texttt{LclDimsUsedIn(ChainA)}\\ &\qquad\textbf{GEN CONSTRAINT: } \bigvee_{\texttt{m} \in \texttt{ChainB}} \texttt{mapEncoding(m)} = 10 + \texttt{l}\end{aligned} \tag{7}$$

Similar constraints are generated for the work group and global domains. All three constraints require that if a parallel dimension of a given domain is used in one map nesting chain, it must also be used in all other chains.

### 4.5 Sequential Map Fusion Heuristic

Perfectly nested sequential maps can always be fused to reduce search space:

$$\begin{aligned}&\forall\texttt{Chain} \in \texttt{MapNestingChains},\ \forall\texttt{m1} \in \texttt{Chain},\ \forall\texttt{m2} \in \texttt{Chain},\\ &\quad\texttt{m2.perfectlyNestedIn(m1)}\\ &\qquad\textbf{GEN CONSTRAINT: } \neg((\texttt{mapEncoding(m1)} = 0) \wedge\\ &\qquad\qquad\qquad\qquad\qquad(\texttt{mapEncoding(m2)} = 0))\end{aligned} \tag{8}$$

This forces all perfectly nested map pairs to be either fused or parallelized. Section 5.5 discusses how this heuristic affects the search.

### 4.6 Synchronizability

Safe parallelization requires that interdependent threads are synchronizable. The following expression is an example where this might be impossible:

$$\begin{aligned}&\texttt{mapLcl}^A\texttt{(1)(mapLcl}^B\texttt{(0)(f) o transpose o mapLcl}^C\texttt{(0)(g)}\\ &\qquad\texttt{) \$ (X: [}T\texttt{]}_n\texttt{)}\end{aligned}$$

transpose introduces an inter-thread dependency between $\texttt{mapLcl}^B$ and $\texttt{mapLcl}^C$, forcing the compiler to insert a barrier between the loops. However, depending on n and the work group size, some threads might perform fewer iterations:

```
for (int i$^A$ = get_local_id(1);
     i$^A$ < n / get_local_size(1);
     i$^A$ += get_local_size(1)) {..}
```

get_local_id(1) and get_local_size(1) are OpenCL built-in primitives that return local thread index and the work group size respectively in the dimension 1. When n is not multiple of the work group size, threads perform differing numbers of iterations. With a barrier inside the loop, some threads get blocked indefinitely.

In Lift, the synchronizability condition is enforced through a compiler check just before code generation. Barrier locations are determined by analyzing loop bounds, computed using tuned parameters such as tile sizes and work group dimensions. Modeling synchronizability as a constraint before tuning would require estimating barrier locations conservatively. By sacrificing early detection in this case, better search coverage is achieved.

## 5 Evaluation

### 5.1 Experimental Methodology

Convolutional layers of the VGG-16 are expressed in Lift. We compare against TVM and the ARM Compute library on the HiSilicon Kirin 970 SoC embedded GPU (ARM Mali-G72 with 12 cores). GPU frequency is fixed to 767MHz (maximum). Each inference is performed over one image. The search and Lift compilation are timed on an Intel Xeon E5-2630v3 with 8GB RAM.

**5.1.1 Lift.** Convolution is automatically parallelized by constructing a search space through constraint generation using the Choco-solver library [17] v4.10.1, which is a good fit since it supports nonlinear integer constraints. Values of the tuning parameters such as tile sizes and thread configuration are then chosen heuristically to saturate compute cores and registers. Vectorization is applied wherever possible by analyzing array indices. Each low-level expression is compiled into a C++ host code and a set of OpenCL kernels. The best candidate is chosen through randomized exploration based on time and memory consumption measurements.

For the run time measurements, we use our own OpenCL profiler — a wrapper that intercepts cl_event instances raised on the start and finish of the OpenCL kernel executions. Each candidate is run 3 times and the median value is reported. Functional correctness is verified against a manually written C implementation.

The parallel mapping search times include the penalty of evaluating the candidates that satisfy the constraints but fail the extra ad-hoc checks. We use Lift compiler memory allocation reports to calculate the exact memory consumption of the generated programs.

**5.1.2 ARM Compute Library.** ARM Compute (v19.02) is used to produce OpenCL implementations of VGG-16 convolutional layers. The implementations are tuned using the ARM Compute auto-tuner. The library produces both direct and GEMM-based implementations.

The performance is profiled by intercepting the OpenCL events. Memory consumption is calculated manually based on the stencil and GEMM algorithms.

**5.1.3 TVM.** TVM is chosen as the comparison code generator since it generally offers better performance than competing frameworks [11]. We use TVM v0.6, built with OpenCL support generated using LLVM version 4.0.0. For a fair comparison, the Winograd strategy is disabled in the Python wrapper of TVM. Winograd implementations incur a loss of precision and are therefore

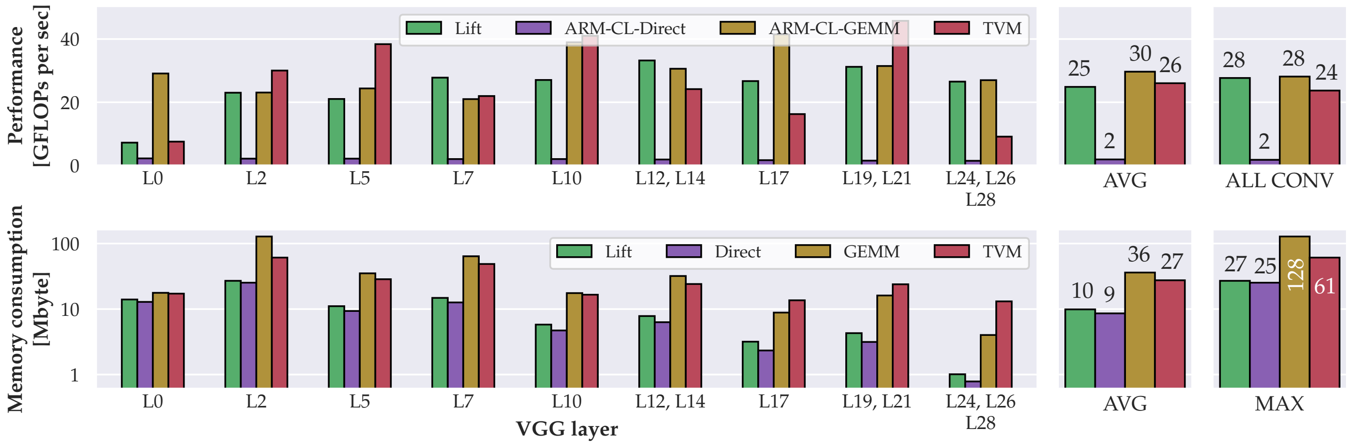


**Figure 4.** Performance and memory consumption comparison of Lift-generated kernels versus the direct and GEMM-based convolution methods in ARM Compute library and TVM-generated kernels on VGG-16 convolutional layers

not semantically equivalent to convolution, while LIFT preserves the semantics of the original problem. We used TVM's preferred convolution method for the Mali GPU: Spatial Pack Convolution [25], which applies im2col and GEMM on the tiled input.

The TVM auto-tuner, with the GATuner, explores 1000 candidates discarding slower implementations that take longer than 100 ms to finish. A median of 30 trials per candidate is recorded. Convolutional layers are constructed using the `Relay` operation, `conv2d`, with all weights and input being represented in `float32` format. TVM compiler optimization is set to level 3. We use TVM auto-tuner reports to collect run time measurements, and we calculate memory consumption based on the intercepted OpenCL memory allocation calls.

### 5.2 Results

Performance and memory consumption of the best implementations are provided in Figure 4. The performance of each implementation is measured in floating-point operations (FLOPs) per second. To obtain framework performance on an individual layer, the theoretical number of FLOPs required by the layer configuration is divided by the time spent computing layer outputs. Compared with the pure run time, the chosen metric is normalized across layers of varying sizes. Within the 3 trials, the first one often suffers the GPU "warmup" overhead; the variance among the remaining runs is within 1%.

Figure 4 also provides average per-layer performance and memory consumption. Performance across all convolutional layers (ALL CONV) takes into account duplicate layer configurations; it is calculated by dividing the total number of FLOPs of all convolutional layers by the total run time. The end-to-end run time of VGG can be inferred from ALL CONV since most of the time is spent computing convolution.

**Table 4.** Breakdown of parallelization constraints generated by the LIFT from the mid-level convolution lambda

| Constraint | # of instances |
|---|---|
| Private memory | 32 |
| Shared memory | 13 |
| Duplicate scheduling | 34 |
| Local thread indexing dimensionality | 1 |
| Local thread indexing hierarchy | 34 |
| Exhaustive thread indexing | 9 |

On average, LIFT is 17% slower than ARM Compute GEMM method while consuming 3.6× less memory. Across the whole VGG-16, LIFT is on par with ARM Compute GEMM; the maximum memory consumption is 4.7× smaller. LIFT is on par with TVM on the average layer and is slightly faster across the whole VGG-16. Average and maximum memory consumption is 2.7× and 2.3× better respectively. Compared to the theoretical minimum memory footprint achieved by the direct method, LIFT requires only 1 Mbyte more on average.

### 5.3 Parallelization Analysis

LIFT generated 123 parallelization constraints from the mid-level lambda; the breakdown of the constraint instances is provided in Table 4. Of those, most constraints were generated to prevent naive scheduling mistakes such as duplicate scheduling and wrong thread indexing hierarchy. 32 constraints were required to enforce private memory scoping since the mid-level lambda was optimized to use the register memory as much as possible.

Out of 81,000 generated candidates satisfying all constraints, 5% passed the extra compiler checks; 33% were not synchronizable and 62% used too much memory. This suggests that further effort is warranted to model these checks as constraints to accelerate the search further.

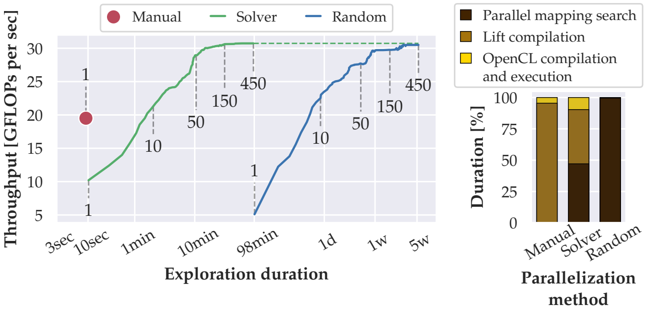


**Figure 5.** Exploration efficiency and the breakdown of time spent in each stage of three parallelization approaches: manual, constraint solver-based and random for VGG-16 layers 19 and 21. The curves are annotated with the numbers of evaluated candidates; the horizontal dotted line shows the projected best throughput

### 5.4 Exploration Efficiency

Figure 5 shows the best throughput achieved as a function of exploration time for the heuristic-based manual approach [14] and the automatic approaches. The constraint solver-based approach outperforms the manual approach after just 88 seconds and reaches peak performance after 95 minutes.

The time breakdown suggests that the solver-based approach spends half the time searching for valid parallel mappings. This increases the time it takes to generate one valid implementation at least twofold compared to the manual approach. The solver-based approach also takes more time in the execution phase due to having to evaluate slow implementations. However, the manual approach requires more time and human expertise to pick a valid parallel mapping.

With the random approach, only 1 out of 49,000 generated candidates satisfies the constraints. In the 98 minutes it takes to produce 1 random valid candidate (with poor performance), the solver-based method already finishes its search achieving the highest throughput. The time spent by the random approach is dominated by the search phase due to the naive search strategy. Both the naive and the solver-based approaches avoid spending time compiling and executing invalid implementations thanks to the parallelization constraints. Despite that and the randomized search in both cases, the solver-based method iteratively reduces the search space during constraint satisfaction through constraint propagation. Parts of the search space are discarded quickly and less time is spent evaluating invalid candidates. The exploration time could be reduced further by adding more constraints expressing parallelization heuristics, synchronizability and memory consumption restrictions, and an objective function.

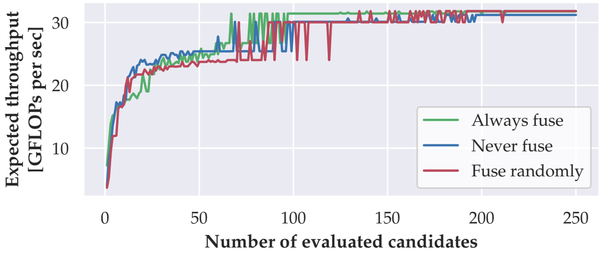


**Figure 6.** Throughput expectancy per number of evaluated candidates with different sequential `map` fusion strategies for VGG-16 convolution layers 19 and 21

### 5.5 Sequential Map Fusion

Throughput expectancy depending on search duration is shown on Figure 6 for three sequential `map` fusion strategies. Expectancy after evaluating $N$ candidates is calculated using a data set of 250 candidates per strategy by uniformly sampling $N$ candidates 100 times and taking a median of 100 maximum throughputs sampled.

The results indicate that always fusing sequential maps does not reduce maximum throughput. The strategy yielded best throughput on VGG-16 layers 19 and 21 after evaluating just 70 candidates compared to 140 for randomized `map` fusion. The difference might be explained by the reduction of search space.

## 6 Related Work

*PPCG* [23], *LetSee* [16], *Tensor Comprehensions* [22], *PlaidML* [24] and *Tiramisu* [2] depend on polyhedral compilation to optimize workloads through linear programming, affine transformations and data access optimizations. By reasoning on the level of loops, polyhedral approaches make the scheduling problem harder and have to resort to heuristic parallelization strategies. The functional patterns in LIFT facilitate static analysis through rich representation, which allows generating parallelization constraints and mapping parallelism automatically.

*PetaBricks* [15] depends on the user to specify the algorithm and an auto-tuner to choose the best candidate. While extensible, this approach limits the search to a set of manually provided implementations. LIFT makes the algorithmic choices transparent to the user.

*Futhark* [8] uses a functional IR to facilitate semantics-preserving rewriting. Through `map` interchange, distribution and fusion, it brings as much nested parallelism outwards as possible and parallelizes the outermost loops only. This limitation puts pressure on the programmer to choose which `maps` to interchange outwards. LIFT can parallelize imperfectly nested loops without supervision.

Like Futhark, *Accelerate* [13] focuses on rewriting the AST to expose more parallelism through loop transformations. For scheduling, it relies on template skeletons.

The *TVM* [4] compiler achieves state-of-the-art performance across state-of-the-art DNNs [11]. TVM's *Halide* IR [18] allows users to specify parametric parallel schedules to be explored by an auto-tuner. Performance is highly dependent on the good initial choice of schedules, while Lift explores a larger search space automatically.

Project *Ansor* [26] provides a TVM auto-scheduler that explores the search space of parallel templates. Ansor can generate invalid parallel mappings, resulting in search time wasted on evaluation of bad candidates. Parallel constraints in Lift prune most invalid kernels; the search is further accelerated since the solver quickly discards parts of the space through constraint propagation.

The *Halide Autoscheduler* [1] uses a tree search algorithm to explore the design space of Halide schedules. On GPUs, the search is limited by course-grained heuristics such as assigning outermost loops to thread blocks. While TVM schedules are integrated into the code generator, Lift rewrite rules are independent and extensible.

Lift and *Spiral* [5] are similar in their use of rewrite rules to transform a functional IR. Spiral also uses constraint satisfaction to ensure valid parallelization. The IR focuses on macro operators such as Cartesian product, direct sum and Kronecker product; the latter is used to capture parallelism. Lift operates on more generic low-level primitives such as `map` and `reduce`.

The *partially specified implementations* IR [3] expresses programs using scalar arithmetic operators and iteration dimensions. The compiler optimizes the implementation through constraint satisfaction. Lift IR is more expressive thanks to its data layout transformations such as `transpose`. This allows explicit control over coalescing and creates vectorization opportunities.

# 7 Conclusions

As seen, the Lift IR exposes plenty of parallelism. This paper demonstrates how this parallelism is exploited safely by auto-generating parallelization restrictions from the rich representation of the program. This shows that the IR can be leveraged to both increase the design space through extensive rewriting and to prune the invalid candidates from the space.

# Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. This work has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF). We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

# A Artifact Appendix

## A.1 Abstract

This artifact contains the parallel code generator Lift which is described in the CC 2022 paper *Mapping Parallelism in a Functional IR through Constraint Satisfaction.* Furthermore, this artifact contains the Docker image, scripts and best-found convolution kernels required to reproduce the performance and search efficiency results presented in the paper. To validate the results, build Lift, run the best-found VGG-16 convolution kernels on a Mali GPU board, and, finally use the plotting script to reproduce the results from Figure 4 in the paper. We also provide scripts to perform time-intensive parallel mapping space exploration and discover the best parallelizations for each layer of VGG-16 for a given Mali GPU board, as well as measure exploration duration to reproduce Figure 5 in the paper.

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

- **Program:** The Lift parallel code generator implemented in Scala
- **Compilation:** With provided scripts
- **Model:** VGG-16 is included as part of the artifact
- **Data set:** Included as part of the artifact
- **Run-time environment:** Linux
- **Hardware:** a host machine to run Lift and a Linux machine with a Mali G72 GPU
- **Metrics:** throughput (GFLOPs per sec), memory consumption (MByte), search duration (sec)
- **Output:** Runtime in CSV files, memory consumption in the generated OpenCL kernels and plots as PDF
- **Experiments:** Git clone; build software; run the OpenCL kernels on the Mali board; observe performance results; run the search; observe search efficiency
- **How much disk space required (approximately)?:** 3 GBytes
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 10 minutes to reproduce Figure 4 with best-found kernels, 30 hours to run the search and reproduce Figures 4 and 5
- **Publicly available:** Yes
- **Code licenses:** MIT
- **Archived:** 10.6084/m9.figshare.19249817

## A.3 Description

### A.3.1 How to access.

The artifact is publicly available and hosted on `gitlab` at https://gitlab.com/naummo/liftpar-cc-2022-artifact.

### A.3.2 Hardware dependencies.

A host Linux machine, and a board with Mali G72 running Linux.

### A.3.3 Software dependencies.

The host machine requires git, git-lfs, wget, curl. For a Docker installation, the Docker engine is required; for a direct installation, the following are required: Python 3+, Texlive LaTeX, Lift, Java 8 SDK, SBT, CMake. The Mali board requires Debian OS with OpenCL support and gcc 7.2.1.

**A.3.4 Data sets.** Datasets are included in the artifact.

**A.3.5 Models.** Models are included in the artifact.

### A.4 Installation

After cloning the repository, a Docker image is provided to set up all requirements on the host machine. For a direct installation, build scripts are provided for Lift, SBT and CMake. An OpenCL device should be set up with a Linux, OpenCL library and gcc. Detailed installation descriptions are given on `gitlab`.

### A.5 Experiment workflow

The provided scripts should be used for plotting the results. Two workflows are suggested: reproducing the main results presented on Figure 4 by rerunning the best-found kernels and performing a time-consuming parallel mapping search to rediscover good kernels naturally. The latter workflow also produces timings required to plot Figure 5. Detailed descriptions for the experiment workflows are provided on the `gitlab` page.

### A.6 Evaluation and expected results

The main results of the artifact evaluation are to reproduce the performance and memory consumption comparison given in Figure 4, and the search efficiency evaluation given in Figure 5. Depending on the host machine we expect the search to take a similar amount of time as reported in the paper.

The reviewers are invited to investigate the implementation of convolution and parallelization constraints in Lift. The `gitlab` page describes how to access Lift expressions, constraint production rules implementation and generated C++ and OpenCL implementations.

### A.7 Experiment customization

The experiment can be customized to find good parallel mappings for a different OpenCL device, a different convolutional neural network or another application. The `gitlab` page describes how to provide the target device specifications and new parallelization constraints to Lift to truncate the search space accordingly.

## References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[3] Ulysse Beaugnon, Basile Clément, Nicolas Tollenaere, and Albert Cohen. 2019. On the Representation of Partially Specified Implementations and its Application to the Optimization of Linear Algebra Kernels on GPU. *arXiv preprint arXiv:1904.03383* (2019).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[5] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[6] The Khronos Group. 2022. OpenCL Reference Pages. https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/

[7] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 100–112.

[8] Troels Henriksen, Niels GW 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*. 556–571.

[9] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7132–7141.

[10] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Not all ops are created equal! *arXiv preprint arXiv:1801.04326* (2018).

[11] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[12] Arm Ltd. 2021. Arm Compute Library. https://developer.arm.com/ip-products/processors/machine-learning/compute-library

[13] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. *ACM SIGPLAN Notices* 48, 9 (2013), 49–60.

[14] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. 2020. Automatic generation of specialized direct convolutions for mobile GPUs. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 41–50.

[15] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2013. Portable performance on heterogeneous architectures. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 431–444.

[16] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. *ACM SIGPLAN Notices* 43, 6 (2008), 90–100.

[17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2016. Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR* 6241 (2016).

[18] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[19] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[20] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 1–10.

[21] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[22] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[23] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[24] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor compilation via the nested polyhedral model. *arXiv preprint arXiv:1903.06498* (2019).

[25] Lanmin Zheng and Tianqi Chen. 2018. Optimizing deep learning workloads on ARM GPU with TVM. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*. 1.

[26] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.