Generating Configurable Hardware from Parallel Patterns

Raghu Prabhakar
Stanford University
raghup17@stanford.edu

David Koeplinger
Stanford University
dkoeplin@stanford.edu

Kevin J. Brown
Stanford University
kjbrown@stanford.edu

HyoukJoong Lee
Stanford University
Google, USA
hyouklee@stanford.edu

Christopher De Sa
Stanford University
cdesa@stanford.edu

Christos Kozyrakis
Stanford University
EPFL
kozyraki@stanford.edu

Kunle Olukotun
Stanford University
kunle@stanford.edu

Abstract
In recent years the computing landscape has seen an increasing shift towards specialized accelerators. Field programmable gate arrays (FPGAs) are particularly promising for the implementation of these accelerators, as they offer significant performance and energy improvements over CPUs for a wide class of applications and are far more flexible than fixed-function ASICs. However, FPGAs are difficult to program. Traditional programming models for reconfigurable logic use low-level hardware description languages like Verilog and VHDL, which have none of the productivity features of modern software languages but produce very efficient designs, and low-level software languages like C and OpenCL coupled with high-level synthesis (HLS) tools that typically produce designs that are far less efficient.

Functional languages with parallel patterns are a better fit for hardware generation because they provide high-level abstractions to programmers with little experience in hardware design and avoid many of the problems faced when generating hardware from imperative languages. In this paper, we identify two important optimizations for using parallel patterns to generate efficient hardware: tiling and metapipelining. We present a general representation of tiled parallel patterns, and provide rules for automatically tiling patterns and generating metapipelines. We demonstrate experimentally that these optimizations result in speedups up to $39.4 \times$ on a set of benchmarks from the data analytics domain.

Keywords
Hardware generation; tiling; metapipelining; parallel patterns; reconfigurable hardware; FPGAs

1. Introduction
The slowdown of Moore’s law and the end of Dennard scaling has forced a radical change in the architectural landscape. Computing systems are becoming increasingly parallel and heterogeneous, relying on larger numbers of cores and specialized accelerators. Field programmable gate arrays (FPGAs) are particularly promising as an acceleration technology, as they can offer performance and energy improvements for a wide class of applications while also providing the reprogrammability and flexibility of software. Applications which exhibit large degrees of spatial and temporal locality and which contain relatively small amounts of control flow, such as those in the image processing [22, 7], financial analytics [31, 17, 53], and scientific computing domains [45, 2, 12, 55], can especially benefit from hardware acceleration with FPGAs. FPGAs have also recently been used to accelerate personal assistant systems [24] and machine learning algorithms like deep belief networks [33, 34].

The performance and energy advantages of FPGAs are now motivating the integration of reconfigurable logic into data center computing infrastructures. Both Microsoft [40] and Baidu [33] have recently announced such systems. These systems have initially been in the form of banks of FPGA accelerators which communicate with CPUs through Infiniband or PCIe [30]. Work is also being done on heterogeneous motherboards with shared CPU-FPGA memory [23]. The recent acquisition of Altera by Intel suggests that systems with tighter, high performance on-chip integration of CPUs and FPGAs are now on the horizon.

The chief limiting factor in the general adoption of FPGAs is that their programming model is currently inaccessible to most software developers. Creating custom accelerator architectures on an FPGA is a complex task, requiring the coordination of large numbers of small, local memories, communication with off-chip memory, and the synchronization of many compute stages. Because of this complexity,
attaining the best performance on FPGAs has traditionally required detailed hardware design using hardware description languages (HDL) like Verilog and VHDL. This low-level programming model has largely limited the creation of efficient custom hardware to experts in digital logic and hardware design.

In the past ten years, FPGA vendors and researchers have attempted to make reconﬁgurable logic more accessible to software programmers with the development of high-level synthesis (HLS) tools, designed to automatically infer register transaction level (RTL) speciﬁcations from higher level software programs. To better tailor these tools to software developers, HLS work has typically focused on imperative languages like C/C++, SystemC, and OpenCL [50]. Unfortunately, there are numerous challenges in inferring hardware from imperative programs. Imperative languages are inherently sequential and effectful. C programs in particular offer a number of challenges in alias analysis and detecting false dependencies [19], typically requiring numerous user annotations to help HLS tools discover parallelism and determine when various hardware structures can be used. Achieving efﬁcient hardware with HLS tools often requires an iterative process to determine which user annotations are necessary, especially for software developers less familiar with the intricacies of hardware design [16].

Functional languages are a much more natural ﬁt for high-level hardware generation as they have limited to no side effects and more naturally express a dataﬂow representation of applications which can be mapped directly to hardware pipelines [6]. Furthermore, the order of operations in functional languages is only deﬁned by data dependencies rather than sequential statement order, exposing signiﬁcant ﬁne-grained parallelism that can be exploited efﬁciently in custom hardware.

Parallel patterns like map and reduce are an increasingly popular extension to functional languages which add semantic knowledge about memory access patterns and inherent data parallelism that is highly exploitable by both software and hardware. Previous work [20, 4] has shown that compilers can utilize parallel patterns to generate C- or OpenCL-based HLS programs and add certain annotations automatically. However, like hand-written HLS, the quality of the generated hardware is still highly variable. Apart from the practical advantage of building on existing tools, generating imperative code from a functional language only to have the HLS tool attempt to re-infer a functional representation of the program is a suboptimal solution because higher-level semantic knowledge in the original program is easily lost. In this paper, we describe a series of compilation steps which automatically generate a low-level, efﬁcient hardware design from an intermediate representation (IR) based on parallel patterns. As seen in Figure 1, these steps fall into two categories: high level parallel pattern transformations (Section 4), and low level analyses and hardware generation optimizations (Section 5).

One of the challenges in generating efﬁcient hardware from high level programs is in handling arbitrarily large data structures. FPGAs have a limited amount of fast local memory and accesses to main memory are expensive in terms of both performance and energy. Loop tiling has been extensively studied as a solution to this problem, as it allows data structures with predictable access patterns to be broken up into ﬁxed size chunks. On FPGAs, these chunks can be stored locally in buffers. Tiling can also increase the reuse of these buffers by reordering computation, thus reducing the number of total accesses to main memory. Previous work on automated tiling transformations has focused almost exclusively on imperative C-like programs with only afﬁne, data-independent memory access patterns. No uniﬁed procedure exists for automatically tiling a functional IR with parallel patterns. In this paper, we outline a novel set of simple transformation rules which can be used to automatically tile parallel patterns. Because these rules rely on pattern matching rather than a mathematical model of the entire program, they can be used even on programs which contain random and data-dependent accesses.

Our tiled intermediate representation exposes memory regions with high data locality, making them ideal candidates to be allocated on-chip. Parallel patterns provide rich semantic information on the nature of the parallel computation at multiple levels of nesting as well as memory access patterns at each level. In this work, we preserve certain semantic properties of memory regions and analyze memory access patterns in order to automatically infer hardware structures like FIFOs, double buffers, and caches. We exploit parallelism at multiple levels by automatically inferring and generating metapipelines, hierarchical pipelines where each stage can itself be composed of pipelines and other parallel constructs. Our code generation approach involves mapping parallel IR constructs to a set of parameterizable hardware templates, where each template exploits a speciﬁc parallel pattern or memory access pattern. These hardware templates are implemented using a low-level Java-based hardware generation language (HGL) called MaxJ.
In this paper we make the following contributions:

- We describe a systematic set of rules for tiling parallel patterns, including a single, general pattern used to tile all patterns with fixed output size. Unlike previous automatic tiling work, these rules are based on pattern matching and therefore do not restrict all memory accesses within the program to be affine.

- We demonstrate a method for automatically inferring complex hardware structures like double buffers, caches, CAMs, and banked BRAMs from a parallel pattern IR. We also show how to automatically generate meta-pipelines, which are a generalization of pipelines that greatly increase design throughput.

- We present experimental results for a set of benchmark applications from the data analytics domain running on an FPGA and show the performance impact of the transformations and hardware templates presented.

2. Related Work

Tiling  

Previous work on automated loop tiling has largely focused on tiling imperative programs using polyhedral analysis [9, 37]. There are many existing tools—such as Pluto [10], PoCC [38], ChiLL [15], and Polly [21]—that use polyhedral analysis to automatically tile and parallelize programs. These tools restrict memory accesses within loops to affine functions of the loop iterators. As a consequence, while they perform well on affine sections of programs, they fail on commonly occurring data-dependent operations like filters and groupBys [8]. In order to handle these operations, recent work has proposed using preprocessing steps which segment programs into affine and non-affine sections prior to running polyhedral analysis tools [51].

While the above work focused on the analysis of imperative programs, our work analyzes functional parallel patterns, which offer a strictly higher-level representation than simple imperative for loops. In this paper, we show that because of the additional semantic information available in patterns like groupBy and filter, parallel patterns can be automatically tiled using simple transformation rules, without the restriction that all memory accesses are purely affine. Little previous work has been done on automated tiling of functional programs composed of arbitrarily nested parallel patterns. Hielscher proposes a set of formal rules for tiling parallel operators map, reduce, and scan in the Parakeet JIT compiler, but these rules can be applied only for a small subset of nesting combinations [25]. Spartan [26] is a runtime system with a set of high-level operators (e.g., map and reduce) on multi-dimensional arrays, which automatically tiles and distributes the arrays in a way that minimizes the communication cost between nodes in cluster environments. In contrast to our work, Spartan operates on a tiled representation for distributed CPU computation and does attempt to optimize performance on individual compute units.

Hardware from high-level languages  

Generating hardware from high-level languages has been widely studied for decades. CHiMPS [41] generates hardware from ANSI C code by mapping each C language construct in a data-flow graph to an HDL block. Kiwi [44] translates a set of C# parallel constructs (e.g., event, monitor, and lock) to corresponding hardware units. Bluespec [3] generates hardware from purely functional descriptions based on Haskell. Chisel [5] is an embedded language in Scala for hardware generation. AutoPilot [54] is a commercial HLS tool that generates hardware from C/C++/SystemC languages. Despite their success in raising the level of abstraction compared to hardware description languages, programmers are still required to write programs at a low-level and express how computations are pipelined and parallelized. Our work abstracts away the implementation details from programmers by using high-level parallel patterns, and applies compiler transformations to automatically pipeline and parallelize operations and exploit on-chip memory for locality.

Existing hardware synthesis tools are limited in their ability to automatically infer and generate coarse-grained pipelines. A traditional software pipelining approach is typically used on innermost loop bodies consisting only of primitive operations. Optimizations like unroll-and-jam, and unroll-and-squash [35] also attempt to exploit pipelined parallelism, but target outer parallel loops with inner sequential loops. To pipeline imperfectly nested loops, some commercial high-level synthesis tools like Vivado [1] unroll all inner loops and then employ traditional software pipelining. Not only does this approach generate needlessly large designs for large benchmarks, it also suffers from long compilation times due to the complexity in scheduling a large number of unrolled instructions. More recent works like ElasticFlow [49] and CGPA [29] generate coarse-grained pipelines using FIFOs in between stages for communication. However, they handle only a restricted form of data access patterns and a restricted number of nesting levels. Our metapipelining technique is more general than previous approaches because: (i) metapipeline stages are decoupled using double buffers, therefore not restricting data access patterns, (ii) metapipelines are easily composed and nested to any number of levels, and (iii) metapipelines can handle dynamic rate mismatches as they use asynchronous handshaking for inter-stage synchronization, thereby obviating the need to calculate static initiation interval as well as knowing loop trip counts ahead of time.

Recent work has explored using polyhedral analysis with HLS to optimize for data locality on FPGAs [39]. Using polyhedral analysis, the compiler is able to promote memory references to on-chip memory and parallelize independent loop iterations with more hardware units. However, the compiler is not able to analyze loops that include non-affine accesses, limiting the coverage of applications that can be generated for hardware. Our work can handle parallel pat-
3. Parallel Patterns

Parallel patterns are becoming a popular programming abstraction for writing high level applications that can still be efficiently mapped to hardware targets such as multicore [32, 36, 46], clusters [18, 52, 26], GPUs [13, 28], and FPGAs [4, 20]. In addition, they have been shown to provide high productivity when implementing applications in a wide variety of domains [48, 42]. We refer to the definitions presented in Figure 2 as the parallel pattern language (PPL).

The definitions on the left represent the atoms in the intermediate language used in our compiler for analysis, optimization, and code generation. The code snippets on the right show common examples of how users typically interact with these patterns in a functional programming language via collections operations and how those examples are represented in PPL. The syntactic structure is essentially the same except that the input domain is inferred from the shape of the input collection. Using explicit indices in the intermediate language allows us to model more user-facing patterns and more input access patterns with fewer internal primitives.

We separate our parallel patterns into two groups. Multidimensional patterns have an arbitrary arity domain and range, but are restricted to a range which is a fixed function of the domain. One-dimensional patterns can have a dynamic output size. All patterns generate output values by applying a function to every index in the domain. Each pattern then merges these values into the final output in a differ-

### Figure 2. Definitions and usage examples of supported parallel patterns.
ent way. The output type $V$ can be a scalar or structure of scalars. We currently do not allow nested arrays, only multi-dimensional arrays. We denote multidimensional array types as $V_{R^k}$, which denotes a tensor of element type $V$ and arity $R$. In Figure 2 subscript $R$ always represents the arity of the output range, and $D$ the arity of the input domain.

Map generates a single element per index, aggregating the results into a fixed-size output collection. Note that the value function can close over an arbitrary number of input collections, and therefore this pattern can represent classic parallel operations like map, zip, and zipWithIndex.

MultiFold is a generalization of a fold which reduces generated values into a specified region of a (potentially) larger accumulator using an associative combine function. The initial value $z$ is required to be an identity element of this function, and must have the same size and shape as the final output. The main function $f$ generates an index specifying the location within the accumulator at which to reduce the generated value. We currently require the generated values to have the same arity as the full accumulator, but they may be of any size up to the size of the accumulator. Note that a traditional fold is the special case of MultiFold where every generated value is the full size of the accumulator. $f$ then converts each index into a function that consumes the specified slice of the current accumulator and returns the new slice. If the pattern’s implementation maintains multiple partial accumulators in parallel, the combine function $c$ reduces them into the final result.

FlatMap generates an arbitrary number of values per index. These values are all concatenated into a flattened output. As the output size can only be determined dynamically, we restrict FlatMap to one-dimensional domains so that dynamically growing the output is easily defined. Note that this primitive also easily expresses a filter.

GroupByFold reduces generated values into one of many buckets where the bucket is selected by generating a key along with each value, i.e., it is a fused version of a groupBy followed by a fold over each bucket. The operation is similar to MultiFold except that the key-space cannot be determined in advance and so the output size is unknown. Therefore we also restrict this operation to one-dimensional domains.

Example Now that we have defined the operations, we will use them to implement k-means clustering as an example application. For reference, first consider k-means implemented using the standard Scala collections operations, as shown in Figure 3. We will use this application as a running example throughout the remainder of this paper, as it exemplifies many of the advantages of using parallel patterns as an abstraction for generating efficient hardware.

```scala
// data to be clustered, size n x d
val points: Array[Array[Float]] = ...

// current centroids, size k x d
val centroids: Array[Array[Float]] = ...

// Assign each point to the closest centroid by grouping
val groupedPoints = points.groupBy(pt1 => pt1)

// Assign current point to the closest centroid
val minDistWithIndex = centroids.zip(pt1).map { case (a,b) => square(a - b) }.sum

// Assign current point to the closest centroid
val newCentroids = groupedPoints.map { case (k,v) => 
  v.reduce((a,b) => a + b)
}.toArray

// Average of points assigned to each centroid
val newCentroids = newCentroids.map { case (k,v) => 
  v.zip(b).map { case (a,b) => a + b }
}.toArray

// Average of points assigned to each centroid
val newCentroids = newCentroids.map { case (k,v) => 
  v.reduce((a,b) => a + b)
}

// data to be clustered
points: Array2D[Float](n,d) // data to be clustered
centroids: Array2D[Float](k,d) // current centroids

// Sum and number of points assigned to each centroid
(sum,counts) = multiFold(points: Array2D[Float](n,d), centroids: Array2D[Float](k,d))

// Assign current point to the closest centroid
val minDistWithIndex = foldWithIndex(points, centroids)

// Assign current point to the closest centroid
val newCentroids = foldWithIndex(points, centroids)

// Average of points assigned to each centroid
val newCentroids = newCentroids.map { case (k,v) => 
  v.reduce((a,b) => a + b)
}

// data to be clustered
points: Array2D[Float](n,d) // data to be clustered
centroids: Array2D[Float](k,d) // current centroids
```

Figure 3. k-means clustering implemented using Scala collections. In Scala, _1 and _2 refer to the first and second value contained within a tuple.
The pattern into a pair of perfectly nested patterns. The outer
strip mining algorithm is defined here
struct unwrapping have already been run.
target-agnostic transformations like fusion, code motion, and
is a PPL representation of a program and that well known
strip mining and interchange. We assume here that our input
pattern tiling method is composed of two transformations:
硬件 blocks. To this end, in this section we describe a
lapping communication with computation using prefetching
expensive and area is limited, our goal is to store a working
chitectures from high level languages is in coping with arbi-
one of the key challenges of generating efficient custom ar-
We have also converted the nested arrays in the Scala exam-
been lowered into PPL and fusion rules have been applied.
We assume a high-level translation from user code to PPL exists
implementation, we use the Delite compiler framework [46] to
produce a view of a subset of the underlying data. In our im-

d =

\text{FlatMap}(d/b)(d)(\text{zeros}(d))\text{\{ }i \rightarrow \\
(\text{\{ }i, \text{acc} \rightarrow \text{Map}(b)(T[m]) \text{\} })\}


\text{GroupByFold}(d/b)(T[z])\text{\{ }i \rightarrow \\
\text{\{ }i, \text{acc} \rightarrow T[c](\text{acc, GroupByFold}(d/b)(T[z]))(T[g])(T[c]) \text{\} })\}

\text{MultiFold}(d/b)(d)(\text{zeros}(d))\text{\{ }i => \\
(\text{\{ }i, \text{acc} \rightarrow \text{GroupByFold}(b)(T[z])\text{\{ }T[g]\text{\} })(T[c]) \text{\} })\text{\} }

\text{GroupByFold}(b)(T[z])(T[h])(T[c])

\text{Map}(b)(T[m])


\text{Table 1. Strip mining transformation rules for parallel patterns defined in Figure 2.}

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Strip Mined Pattern</th>
</tr>
</thead>
</table>
| T[ Map(d)(m) ]                 | = \text{MultiFold}(d/b)(d)(\text{zeros}(d))\text{\{ }i \rightarrow \\
|                                | (\text{\{ }i, \text{acc} \rightarrow \text{Map}(b)(T[m]) \text{\} })\} |
| T[ MultiFold(d)(r)(z)(g)(c) ]  | = \text{MultiFold}(d/b)(d)(T[z])\text{\{ }i \rightarrow \\
|                                | (\text{\{ }i, \text{acc} \rightarrow T[c](\text{acc, MultiFold}(d/b)(T[z])(T[g])(T[c])) \text{\} })\} |
| T[ GroupByFold(d)(z)(h)(c) ]   | = \text{GroupByFold}(d/b)(T[z])\text{\{ }i \rightarrow \\
|                                | \text{GroupByFold}(b)(T[z])(T[h])(T[c])\text{\} } |
| T[ FlatMap(d)(f) ]             | = \text{FlatMap}(d/b)(\text{\{ }i \rightarrow \text{FlatMap}(b)(T[f]) \text{\} }) |

values are computed by averaging all the samples assigned to
each centroid. This process repeats until the centroid values
stop changing. Previous work [43, 11, 14] has shown how
to stage a DSL application like k-means, lowering it into a
parallel pattern IR similar to ours, as well as how to perform
multiple high-level optimizations automatically on the IR.
One of the most important of these optimizations is fusing
patterns together, both vertically (to decrease the reuse distance
between producer-consumer relationships) and hori-
izontally (to eliminate redundant traversals over the same do-
main). Figure 4 shows the structure of k-means after it has
been lowered into PPL and fusion rules have been applied.
We have also converted the nested arrays in the Scala exam-
ple to our multidimensional arrays. This translation requires
the insertion of slice operations in certain locations, which
produce a view of a subset of the underlying data. In our im-
plementation, we use the Delite compiler framework [46] to
stage applications. For the remainder of this paper, we will
assume a high-level translation from user code to PPL exists
and always start from the parallel pattern representation.

4. Pattern Transformations

One of the key challenges of generating efficient custom ar-
chitectures from high level languages is in coping with arbi-
trarily large data structures. Since main memory accesses are
expensive and area is limited, our goal is to store a working
set in the FPGA’s local memory for as long as possible. Ide-
ally, we also want to hide memory transfer latencies by over-
lapping communication with computation using prefetching
hardware blocks. To this end, in this section we describe a
method for automatically tiling parallel patterns to improve
program locality and data reuse. Like classic loop tiling, our
pattern tiling method is composed of two transformations:
strip mining and interchange. We assume here that our input
is a PPL representation of a program and that well known
target-agnostic transformations like fusion, code motion, and
struct unwrapping have already been run.

Strip mining The strip mining algorithm is defined here
using two passes over the IR. The first pass partitions each
pattern’s iteration domain d into tiles of size b by breaking
the pattern into a pair of perfectly nested patterns. The outer
pattern operates over the strided index domain, expressed
here as d/b, while the inner pattern operates on a tile of
size b. For the sake of brevity this notation ignores the case
where b does not perfectly divide d. This case is trivially
solved with the addition of min checks on the domain of the
inner loop. Table 1 gives an overview of the rules used by
transformer (denoted T) to strip mine parallel patterns. In
addition to splitting up the domain, patterns are transformed
by recursively strip mining all functions within that pattern.
Map is strip mined by reducing its domain and range and
nesting it within a MultiFold. Note that the strided MultiFold
writes to each memory location only once. In this case we
indicate the MultiFold’s combination function as unused
with an underscore. As defined in Figure 2, the MultiFold,
GroupByFold, and FlatMap patterns have the property that a
perfectly nested form of a single instance of one of these
patterns is equivalent to a single “flattened” form of that
same pattern. This property allows these patterns to be strip
mined by breaking them up into a set of perfectly nested
patterns of the same type as the original pattern.

The second strip mining pass converts array slices and ac-
cesses with statically predictable access patterns into slices
and accesses of larger, explicitly defined array memory tiles.
We define tiles which have a size statically known to fit on
the FPGA using array copies. Copies generated during strip
mining can then be used to infer buffers during hardware
generation. Array tiles which have overlap, such as those
generated from sliding windows in convolution, are marked
with metadata in the IR as having some reuse factor. Array
copies with reuse have generation rules which minimize the
number of redundant reads to main memory when possible.

Table 2 demonstrates how our rules can be used to strip
mine a set of simple data parallel operations. We use the copy
infix function on arrays to designate array copies in these
examples, using similar syntax as array slice. We assume
in these examples that common subexpression elimination
(CSE) and code motion transformation passes have been run
after strip mining to eliminate duplicate copies and to move
array tiles out of the innermost patterns. In these examples,
strip mining creates tiled copies of input collections that we
can later directly use to infer read buffers.
Pattern interchange  Given an intermediate representation with strip mined nested parallel patterns, we now need to interchange patterns to increase the reuse of newly created data tiles. This can be achieved by moving strided patterns out of unstrided patterns. However, as with imperative loops, it is not sound to arbitrarily change the order of nested parallel patterns. We use two rules for pattern interchange adapted from a previously established Collect-Reduce reordering rule for computation on clusters [11]. These rules both match on the special case of MultiFold where every iteration updates the entire accumulator, which we refer to here as a fold. The first interchange rule defines how to move a scalar, strided fold out of an unstrided Map, transforming the nested loop into a strided fold of a Map. Note that this also changes the combination function of the fold into a Map. The second rule is the inverse of the first, allowing us to reorder a strided MultiFold with no reduction function (i.e. the outer pattern of a tiled Map) out of an unstrided fold. This creates a strided MultiFold of a scalar fold. We apply these two rules whenever possible to increase the reuse of tiled inputs.

Imperfectly nested parallel patterns commonly occur either due to the way the original user program was structured or as a result of aggressive vertical fusion run prior to tiling. Interchange on imperfectly nested patterns requires splitting patterns into perfectly nested sections. However, splitting and reordering trades temporal locality of intermediate values for increased reuse of data tiles. In hardware, this can involve creating more main memory reads or larger on-chip buffers for intermediate results so that less reads need to be done for input and output data. This tradeoff between memory reads and increased buffer usage requires more complex cost modeling. We use a simple heuristic to determine...
whether to split fused loops: we split and interchange patterns only when the intermediate result created after splitting and interchanging is statically known to fit on the FPGA. This handles the simple case where the FPGA has unused on-chip buffers and allocating more on-chip memory guarantees a decrease in the number of main memory reads. Future work will examine ways to statically model the tradeoff between main memory accesses and local buffers near 100% on-chip memory utilization.

Table 3 shows a simple example of the application of our pattern interchange rules on matrix multiplication. We assume here that code motion has been run again after pattern interchange has completed. In matrix multiplication, we inter- change the perfectly nested strided MultiFold and the un- strored Map. This ordering increases the reuse of the copied tile of matrix $y$ and changes the scalar reduction into a tile- wise reduction. Note that the partial result calculation and the inner reduction can now be vertically fused.

**Discussion** The rules we outline here for automatic tiling of parallel patterns are target-agnostic. However, tile copies should only be made explicit for devices with scratchpad memory. Architectures with hierarchical memory systems effectively maintain views of subsections of memory automatically through caching, making explicit copies on these architectures a waste of both compute cycles and memory.

We currently require the user to explicitly specify tile sizes for all dimensions which require tiling. In future work, tile sizes for all pattern dimensions will instead be determined by the compiler through automated tile size selection using modeling and design space exploration.

**Example** We conclude this section with a complete example of tiling the $k$-means clustering algorithm, starting from the fused representation shown in Figure 4. We assume here that we wish to tile the number of input points, $n$, with tile size $b_0$ and the number of clusters, $k$, with tile size $b_1$ but not the number of dimensions, $d$. This is representative of machine learning classification problems where the number of input points and number of labels is large, but the number of features for each point is relatively small.

Figure 5 gives a comparison of the $k$-means clustering algorithm after strip mining and after pattern interchange. During strip mining, we create tiles for both the $points$ and $centroids$ arrays, which helps us to take advantage of main memory burst reads. However, in the strip mined version, we still fully calculate the closest centroid for each point. This requires the entirety of $centroids$ to be read for each point. We increase the reuse of each tile of $centroids$ by first splitting the calculation of the closest centroid label from the MultiFold (Figure 5a, line 5). The iteration over the centroids tile is then perfectly nested within the iteration over the points. Interchanging these two iterations allows us to reuse the centroids tile across points, thus decreasing the total number of main memory reads for this array by a factor of $b_0$. This decrease comes at the expense of changing the intermediate (distance, label) pair for a single point to a set of intermediate pairs for an entire tile of points. Since the created intermediate result has size $2b_0$, we statically determine that this is an advantageous tradeoff and use the split and interchanged form of the algorithm.

## 5. Hardware Generation

In this section, we describe how the tiled intermediate representation is translated into an efficient FPGA design. FPGAs are composed of various logic, register, and memory resources. These resources are typically configured for a specific hardware design using a hardware description language (HDL) that is translated into an FPGA configuration file. Our approach to FPGA hardware generation translates our parallel pattern IR into MaxJ, a Java-based hardware generation language (HGL), which is in turn used to generate an HDL. This is simpler than generating HDL directly because MaxJ performs tasks such as automatic pipelining of innermost loops and other low-level hardware optimizations.

Hardware generation follows a template-based approach. We analyze the structure of the parallel patterns in the IR to determine the correct template to translate the pattern to hardware. Table 4 lists the templates and their corresponding IR constructs in three classes: memories, pipelined execution units, and state machine controllers. $Buffer$, $Double buffer$, and $Cache$ are different on-chip memory templates intended to capture both regular and data-dependent access patterns. In particular, the $double buffer$ template is used to decouple execution stages and support dynamic rate mismatch between producer and consumer stages. Templates labeled as $Pipelined Execution Units$ are used to support different kinds of innermost parallel patterns, as described in Table 4. The $Controller$ templates implement a specific form of control flow using asynchronous handshaking signals. The $Sequential$, $Parallel$, and $Metapipeline$ controllers all orchestrate execution of a list of templates; $Sequential$ enforces linear execution order, $Parallel$ enforces parallel execution with a barrier at the end, and $MetaPipeline$ enforces pipelined execution. $Tile Memory$ controllers correspond to off-chip memory channels that load tiles of data into one of the on-chip memory templates. Each template can be composed with other templates. For example, a $Metapipeline controller$ could be composed of multiple $Parallel controllers$, each of which could contain pipelined $Vector$ or $Tree reduction$ units. We next describe the key features in the IR which we use to infer each of these template classes.

**Memory Allocation** Generating efficient FPGA hardware requires effective usage of on-chip memories (buffers). Prior to generating MaxJ, we run an analysis pass to allocate buffers for arrays based on data access patterns and size. All arrays with statically known sizes, such as array copies generated in the tiling transformation described in Section 4, are assigned to buffers. Dynamically sized arrays are kept in main memory and we generate caches for any non-affine ac-
(a) Strip mined \(k\)-means in PPL.

(b) Pattern Interchanged \(k\)-means in PPL.

(c) Pseudocode description of strip mined \(k\)-means.

(d) Pseudocode description of pattern interchanged \(k\)-means.

(e) Minimum number of words read from main memory and on-chip storage for data structures within \(k\)-means clustering after each IR transformation.

Figure 5. Full tiling example for \(k\)-means clustering, starting from the fused representation in Figure 4, using tile sizes of \(b_0\) and \(b_1\) for the number of points \(n\) and the number of clusters \(k\). The number of features \(d\) is not tiled in this example.
cesses to these arrays. We also track each memory’s readers and writers and use this information to instantiate a template with the appropriate word width and number of ports.

**Pipeline Execution Units** We generate parallelized and pipelined hardware when parallel patterns compute with scalar values, as occurs for the innermost patterns. We implemented templates for each pipelined execution unit in Table 4 using MaxJ language constructs, and instantiate each template with the proper parameters (e.g., data type, vector length) associated with the parallel pattern. The MaxJ compiler applies low-level hardware optimizations such as vectorization, code scheduling, and fine-grained pipelining, and generates efficient hardware. For example, we instantiate a reduction tree for a MultiFold over an array of scalar values, which is automatically pipelined by the MaxJ compiler.

**Metapipelining** To generate high performance hardware from parallel patterns, it is insufficient to exploit only a single level of parallelism. However, exploiting nested parallelism requires mechanisms to orchestrate the flow of data through multiple pipeline stages while also exploiting parallelism at each stage of execution, creating a hierarchy of pipelines, or metapipeline. This is in contrast to traditional HLS tools which require inner patterns to have a static size and be completely unrolled in order to generate a flat pipeline containing both the inner and outer patterns.

We create metapipeline schedules by first performing a topological sort on the IR of the body of the current parallel pattern. The result is a list of stages, where each stage contains a list of patterns which can be run concurrently. Exploiting the pattern’s semantic information, we then optimize the metapipeline schedule by removing unnecessary memory transfers and redundant computations. For instance, if the output memory region of the pattern has been assigned to a buffer, we do not generate unnecessary writes to main memory.

As another example, our functional representation of tiled parallel patterns can sometimes create redundant accumulation functions, e.g., in cases where a MultiFold is tiled into a nested MultiFold. During scheduling we identify this redundancy and emit a single copy of the accumulator, removing the unnecessary intermediate buffer. Finally, in cases where the accumulator of a MultiFold cannot completely fit on-chip, we add a special forwarding path between the stages containing the accumulator. This optimization avoids redundant writes to memory and reuses the current tile. Once we have a final schedule for the metapipeline, we promote every output buffer in each stage to a double buffer to avoid write after read (WAR) hazards between metapipeline stages.

**Example** Figure 6 shows a block diagram of the hardware generated for the k-means application. For simplicity, this diagram shows the case where the centroids array completely fits on-chip, meaning we do not tile either the number of clusters k or the number of features d. The generated hardware contains three sequential steps. The first step (Pipe 0) preloads the entire centroids array into a buffer. The second step (Metapipeline A) is a metapipeline which consists of three stages with double buffers to manage communication between the stages. These three stages directly correspond to the three main sections of the MultiFold (Figure 4, line 5) used to sum and count the input points as grouped by their closest centroid. The first stage (Pipe 1) loads a tile of the points array onto the FPGA. Note that this stage is double buffered to enable hardware prefetching. The second stage (Pipe 2) computes the index of the closest centroid using vector compute blocks and a scalar reduction tree. The third stage (Pipe 3 and Pipe 4) increments the count for this minimum index and adds the current point to the corresponding location in the buffer allocated for the new centroids. The third step (Metapipeline B) corresponds with the second outermost parallel pattern in the k-means application. This step streams through the point sums and the centroid counts, dividing each sum by its corresponding count. The resulting new centroids are then written back to main memory using a tile store unit for further use on the CPU.

Our automatically generated hardware design for the core computation of k-means is very similar to the manually op-

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
<th>IR Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memories</td>
<td>Buffer</td>
<td>On-chip scratchpad memory</td>
</tr>
<tr>
<td></td>
<td>Double buffer</td>
<td>Buffer coupling two stages in a metapipeline</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
<td>Tagged memory to exploit locality in random memory access patterns</td>
</tr>
<tr>
<td>Pipelined Execution Units</td>
<td>Reduction tree</td>
<td>Parallel reduction of associative operations</td>
</tr>
<tr>
<td></td>
<td>Parallel PIFO</td>
<td>Used to buffer ordered outputs of dynamic size</td>
</tr>
<tr>
<td>Controllers</td>
<td>Sequential Controller</td>
<td>Controller which coordinates sequential execution</td>
</tr>
<tr>
<td></td>
<td>Parallel Controller</td>
<td>Task parallel controller. Simultaneously starts all member modules when enabled, signals done when all members finish</td>
</tr>
<tr>
<td></td>
<td>Metapipeline Controller</td>
<td>Controller which coordinates execution of nested parallel patterns in a pipelined fashion</td>
</tr>
<tr>
<td></td>
<td>Tile memory</td>
<td>Memory command generator to fetch tiles of data from off-chip memory</td>
</tr>
</tbody>
</table>

Table 4. Hardware templates used in hardware code generation.
We run each generated design on an Altera Stratix V FPGA on a Max4 Maia board. The Maia board contains 48 GB of DDR3 DRAM with a maximum bandwidth of 76.8 GB/s. The area numbers given in this section are obtained by the hardware design reports completion. Final running times were calculated as an arithmetic mean of five individual runs to account for small runtime variations in main memory accesses and Maxeler’s device driver stack.

We implement our transformation and hardware generation steps in an existing compiler framework called Delite [46]. We write each of our benchmark applications in OptiML [47], a high level, domain specific language embedded in Scala for machine learning. We then compile each of these applications with the modified Delite compiler. During compilation, applications are staged, translating them into PPL representations. The compiler then performs the tiling transformations and hardware optimizations described in Sections 4 and 5 before generating MaxJ hardware designs. We then use the Maxeler MaxCompiler toolchain to generate an FPGA configuration bitstream from our generated MaxJ.

6. Evaluation

We evaluate our approach to hardware generation described in Sections 4 and 5 by comparing the performance and area utilization of the FPGA implementations of a set of data analytic benchmarks. We focus our investigation on the relative improvements that tiling and metapipelining provide over hardware designs that do not have these features.

6.1 Methodology

The benchmarks used in our evaluation are summarized in Table 5. We choose to study vector outer product, matrix row summation, and matrix multiplication as these exemplify many commonly occurring access patterns in the machine learning domain. TPC-H Query 6 is a data querying application which reads a table of purchase records, filtering all records which match a given predicate. It then computes the sum of a product of two columns in the filtered records. Logistic regression is a binary discriminative classification algorithm that uses the sigmoid function in the calculation of predictions. Gaussian discriminant analysis (GDA) is a classification algorithm which models the distribution of each class as a multivariate Gaussian. Black-Scholes is a financial analytics application for option pricing. k-means clustering groups a set of input points by iteratively calculating the $k$ best cluster centroids. In our implementations, all of these benchmarks operate on single precision, floating point data.

We implement our transformation and hardware generation steps in an existing compiler framework called Delite [46]. We write each of our benchmark applications in OptiML [47], a high level, domain specific language embedded in Scala for machine learning. We then compile each of these applications with the modified Delite compiler. During compilation, applications are staged, translating them into PPL representations. The compiler then performs the tiling transformations and hardware optimizations described in Sections 4 and 5 before generating MaxJ hardware designs. We then use the Maxeler MaxCompiler toolchain to generate an FPGA configuration bitstream from our generated MaxJ. We use the Maxeler runtime layer to manage communication with the FPGA from the host CPU. We measure the running times of these designs starting after input data has been copied to the FPGA’s DRAM and ending when the hardware design reports completion. Final running times were calculated as an arithmetic mean of five individual run times to account for small runtime variations in main memory accesses and Maxeler’s device driver stack.

Table 5. Evaluation benchmarks with major collections operations used by Scala implementation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Collections Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>outerprod</td>
<td>Vector outer product</td>
<td>map</td>
</tr>
<tr>
<td>sumrows</td>
<td>Summation through matrix rows</td>
<td>map, reduce</td>
</tr>
<tr>
<td>gemm</td>
<td>Matrix multiplication</td>
<td>map, reduce</td>
</tr>
<tr>
<td>tpchq6</td>
<td>TPC-H Query 6</td>
<td>filter, reduce</td>
</tr>
<tr>
<td>logreg</td>
<td>Logistic regression</td>
<td>map, reduce</td>
</tr>
<tr>
<td>gda</td>
<td>Gaussian discriminant analysis</td>
<td>map, filter, reduce</td>
</tr>
<tr>
<td>blackscholes</td>
<td>Black-Scholes option pricing</td>
<td>map</td>
</tr>
<tr>
<td>kmeans</td>
<td>$k$-means clustering</td>
<td>map, groupBy, reduce</td>
</tr>
</tbody>
</table>

Figure 6. Hardware generated for the $k$-means application.
tained from synthesis reports provided by Altera’s logic synthesis toolchain. Area utilization is reported under three categories: Logic utilization (denoted “logic”), flip flop usage (“FF”), and on-chip memory usage (“mem”).

6.2 Experiments

The baseline for each benchmark is an optimized hardware design implemented using MaxJ. The baseline designs were manually tuned after automatic generation and are representative of optimizations done by state-of-the-art high-level synthesis tools. In particular, each baseline design exploits data and pipelined parallelism within patterns where possible. Pipelined parallelism is exploited for patterns that operate on scalars. Our baseline design exploits locality at the level of a single DRAM burst, which on the MAX4 MAIA board is 384 bytes. To isolate the effects of the amount of parallelism in our comparison, we keep the innermost pattern parallelism factor constant between the baseline design and our optimized versions for each benchmark.

We evaluate our approach against the baseline by generating two hardware configurations per benchmark: a configuration with tiling but no metapipelining, and a configuration with both tiling and metapipelining optimizations enabled.

Impact of tiling alone Figure 7 shows the obtained speedups as well as relative on-chip resource usages for each of benchmarks. As can be seen, most benchmarks in our suite show significant speedup when tiling transformations are enabled. Benchmarks like sumrows and gemm benefit from inherent locality in their memory accesses. For gemm, our automatically generated code achieves a speedup of 4× speedup over the baseline for a marginal increase of about 10% on-chip memory usage.

Benchmarks outerprod and tpchq6 do not show a significant difference with our tiling transformations over the baseline. This is because both outerprod and tpchq6 are both memory-bound benchmarks. Tpchq6 streams through the input once without reuse, and streaming data input is already exploited in our baseline design. Blackscholes has a similar data access pattern as tpchq6, due to which it achieves a speedup similar to that of tpchq6. Hence tiling does not provide any additional benefit. Most of the locality in logreg is already captured at burst-level granularity by our baseline. As a result, logreg achieves a modest speedup of 1.8× over the baseline due to tiling. The core compute pipeline in outerprod is memory-bound at the stage writing results to DRAM, which cannot be addressed using tiling. Despite the futility of tiling in terms of performance, tiling outerprod has a noticeable increase in memory utilization as the intermediate result varies as the square of the tile size.

In kmeans and gda, some of the input data structures are small enough that they can be held in on-chip memory. This completely eliminates accesses to off-chip memory, leading to dramatic speedups of 13.4× and 15.5× respectively with our tiling transformations. gda uses more on-chip memory to store intermediate data. On the other hand, the tiled version kmeans utilizes less on-chip memory resources. This is because the baseline for kmeans instantiates multiple load and store units, each of which creates several control structures in order to read and write data from DRAM. Each of these control structures includes address and data streams, which require several on-chip buffers. By tiling, we require a smaller number of load and store units.

Impact of metapipelining The second speedup bar in Figure 7 shows the benefits of metapipelining. Metapipelines increase throughput at the expense of additional on-chip memory used for double buffers. Metapipelining overlaps design compute with data transfer and helps to hide the cost of the slower stage. Benchmarks like gemm and sumrows naturally
benefit from metapipelining because the memory transfer time is completely overlapped with the compute, resulting in speedups of $6.3 \times$ and $11.5 \times$ respectively. Metapipelining also exploits overlap in streaming benchmarks like tpcq6 and blackscholes, where the input data is fetched and stored simultaneously with the core computation.

The speedup due to metapipelining is largely determined by balancing between stages. Stages with roughly equal number of cycles benefit the most, as this achieves the most overlap. Unbalanced stages are limited by the slowest stage, thus limiting performance. We observe this behavior in outerprod, where the metapipeline is bottlenecked by the stage writing results back to DRAM. The metapipeline in logreg is bottlenecked at the stage performing dot products of all the points in the input tile with the theta vector. As we only parallelize the innermost parallel pattern in this work, only a single dot product is produced at a time, even though the dot product itself is executed in parallel across the point dimensions. On the other hand, applications like gda, kmeans and sumrows greatly benefit from metapipelining. In particular, gda naturally maps to nested metapielines that are well-balanced. The stage loading the input tile overlaps execution with the stage computing the output tile and the stage storing the output tile. The stage computing the output tile is also a metapipeline where the stages perform vector subtraction, vector outer product and accumulation. We parallelize the vector outer product stage as it is the most compute-heavy part of the algorithm; parallelizing the vector outer product enables the metapipeline to achieve greater throughput. This yields an overall speedup of $39.4 \times$ over the baseline.

7. Conclusion

In this paper, we introduced a set of compilation steps necessary to produce an efficient FPGA hardware design from an intermediate representation composed of nested parallel patterns. We described a set of simple transformation rules which can be used to automatically tile parallel patterns which exploit semantic information inherent within these patterns and which place fewer restrictions on the program’s memory accesses than previous work. We then presented a set of analysis and generation steps which can be used to automatically infer optimized hardware designs with metapipelining. Finally, we presented experimental results for a set of benchmarks in the machine learning and data querying domains which show that these compilation steps provide performance improvements of up to $39.4 \times$ with a minimal impact on FPGA resource usage.

Acknowledgments

The authors thank Maxeler Technologies for their assistance with this paper and Jacob Bower for his help running experiments. They also thank the reviewers for their comments and suggestions. This work is supported by DARPA Contract-Air Force FA8750-12-2-0335; Army Contract AH-PCRC W911NF-07-2-0027-1; NSF Grants IIS-1247701, CCF-1111943, CCF-1337375, and SHF-1408911; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Huawei, Intel, NVIDIA, SAP Labs. Authors acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References


