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

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872415

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,

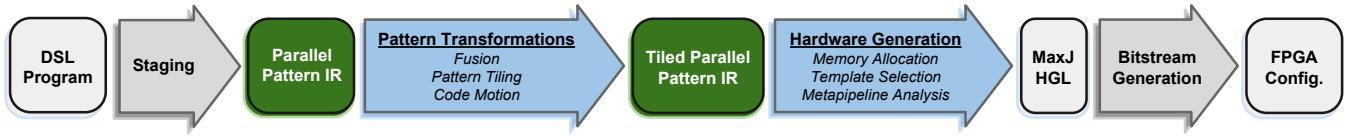

Figure 1. System diagram

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 information 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 *metapipelines*, 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 only affine functions of the loop iterators. As a consequence, while they perform well on affine sections of programs, they fail on even simple, commonly occurring data-dependent operations such as *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 dataflow 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

Parallel Pattern Definition		High Level Language Example	PPL Example
Multidimensional			
`Map(d)(m)`	: V_D	*// Size s vector multiplied by 2* `x.map{ e => 2*e }` *// Addition of two size s vectors* `x.zip(y){ (a,b) => a + b }`	`map(s){i => 2*x(i) }` `map(s){i => x(i) + y(i) }`
`MultiFold(d)(r)(z)(f)(c)`	: V_R	*// Reduction of vector of s elements* `x.fold(1){ (a,b) => a * b }` *// Row summation in s x t matrix* `x.map{ row =>` `  row.fold(0){ (a,b) => a + b }` `}`	`multiFold(s)(1)(1){ i =>` `  (0, acc => acc + x(i))` `}{ (a,b) => a + b }` `multiFold(s,t)(r)(zeros(s)){ (i,j) =>` `  (i, acc => acc + x(i,j) )` `}{ (a,b) => map(s){ i => a(i) + b(i) } }`
One-dimensional			
`FlatMap(d)(n)`	: V_1	*// Filter positives from s elements* `x.flatMap{ e =>` `  if (e > 0) [e] else [] }`	`flatMap(s){ i =>` `  if (x(i) > 0) [x(i)] else [] }`
`GroupByFold(d)(z)(g)(c)`	: $(K,V)_1$	*// Histogram with bin width 10* `x.groupByFold(0){ e =>` `  (e/10, 1)` `}{ (a,b) => a + b }`	`groupByFold(s)(0){ i =>` `  (x(i)/10, acc => acc + 1)` `}{ (a,b) => a + b }`

User-defined Values

d : Integer_D	input domain	m : Index_D => V	value function
r : Integer_R	output range	n : Index => V_1	multi-value function
z : V_R	init accumulator	f : Index_D => (Index_R, V_R => V_R)	(location, value) function
c : (V_R,V_R) => V_R	combine accumulator	g : Index => (K, V => V)$_1$	(key, value) function

Figure 2. Definitions and usage examples of supported parallel patterns.

accesses, limiting the coverage of applications that can be generated for hardware. Our work can handle parallel patterns with non-affine accesses by inferring required hardware blocks (e.g., FIFOs and CAMs) for non-affine accesses, while aggressively using on-chip memory for affine parts.

As high-level parallel patterns become increasingly popular to overcome the shortcomings of C based languages, researchers have recently studied generating hardware from functional parallel patterns. Lime [4] embeds high-level computational patterns (e.g., *map*, *reduce*, *split*, and *join*) in Java and automatically targets CPUs, GPUs, and FPGAs without modifying the code. Our compiler manages a broader set of parallel patterns (e.g., *groupBy*) and applies transformations even when patterns are nested, which is common in a large number of real-world applications. Recent work has explored targeting nested parallel patterns to FPGAs [20]. By exploiting the access patterns of nested patterns to store sequential memory accesses to on-chip memory and parallelizing the computation with strip-mining, the compiler can generate hardware that efficiently utilizes memory bandwidth. However, the compiler does not automatically tile patterns for data locality or implement metapipelines for nested parallel patterns, which we show are essential components for generating efficient hardware. Overall, our work is the first to show a complete method for automatically tiling parallel patterns to improve locality for individual compute units and a process for inferring hardware metapipelines from nested parallel patterns.

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 complicated 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

```
1  //data to be clustered, size n x d
2  val points: Array[Array[Float]] = ...
3
4  // current centroids, size k x d
5  val centroids: Array[Array[Float]] = ...
6
7  // Assign each point to the closest centroid by grouping
8  val groupedPoints = points.groupBy { pt1 =>
9    // Assign current point to the closest centroid
10   val minDistWithIndex = centroids.map { pt2 =>
11     pt1.zip(pt2).map { case (a,b) => square(a - b) }.sum
12   }.zipWithIndex.minBy(p => p._1)
13   minDistWithIndex._2
14 }
15
16 // Average of points assigned to each centroid
17 val newCentroids = groupedPoints.map { case (k,v) =>
18   v.reduce { (a,b) =>
19     a.zip(b).map { case (x,y) => x + y }
20   }.map { e => e / v.length }
21 }.toArray
```

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.

then merges these values into the final output in a different way. The output type V can be a scalar or structure of scalars. We currently do not allow nested arrays, only multidimensional arrays. We denote multidimensional array types as V_R, 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 is general enough to 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 is similar to *Map* except that it can generate an arbitrary number of values per index. These values are then all concatenated into a flattened output. The output size can

```
1  points: Array2D[Float](n,d) // data to be clustered
2  centroids: Array2D[Float](k,d) // current centroids
3
4  // Sum and number of points assigned to each centroid
5  (sums,counts) = multiFold(n)((k,d),k)(zeros((k,d),k)){ i =>
6    pt1 = points.slice(i, *)
7    // Assign current point to the closest centroid
8    minDistWithIndex = fold(k)((max, -1)){ j =>
9      pt2 = centroids.slice(j, *)
10     dist = fold(d)(0){ p =>
11       acc => acc + square(pt1(p) - pt2(p))
12     }{ (a,b) => a + b }
13     acc => if (acc._1 < dist) acc else (dist, j)
14   }{ (a,b) => if (a._1 < b._1) a else b }
15
16   minDistIndex = minDistWithIndex._2
17   sumFunc = ((minDistIndex, 0), acc => {
18     pt = points.slice(i, *)
19     map(d){ j => acc(j) + pt(j) }
20   })
21   countFunc = (minDistIndex, acc => acc + 1)
22
23   (sumFunc, countFunc)
24 }{ (a,b) => {
25   pt = map(k,d){ (i,j) => a._1(i,j) + b._1(i,j) }
26   count = map(k){ i => a._2(i) + b._2(i) }
27   (pt, count)
28 } }
29
30 // Average assigned points to compute new centroids
31 newCentroids = map(k,d){ (i,j) =>
32   sums(i,j) / counts(i)
33 }
```

Figure 4. k-means clustering represented using the parallel patterns in Figure 2 after fusion and code motion.

only be determined dynamically and therefore we restrict the operation 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. k-means consumes a set of n sample points of dimensionality d and attempts to cluster those points by finding the k best cluster centroids for the samples. This is achieved by iteratively refining the centroid values. (We show only one iteration in Figure 3 for simplicity.) First, every sample point is assigned

Pattern		Strip Mined Pattern
`T[ Map(d)(m) ]`	=	`MultiFold(d/b)(d)(zeros(d)){ i =>` `  (i, acc => Map(b)(T[m]) )` `}(_)`
`T[ MultiFold(d)(r)(z)(g)(c) ]`	=	`MultiFold(d/b)(r)(T[z]){ i =>` `  (i, acc => T[c](acc, MultiFold(b)(r)(T[z])(T[g])(T[c])) )` `}(T[c])`
`T[ GroupByFold(d)(z)(h)(c) ]`	=	`GroupByFold(d/b)(T[z]){ i =>` `  GroupByFold(b)(T[z])(T[h])(T[c])` `}(T[c])`
`T[ FlatMap(d)(f) ]`	=	`FlatMap(d/b){i => FlatMap(b)(T[f]) }`

Table 1. Strip mining transformation rules for parallel patterns defined in Figure 2.

to the closest current centroid by computing the distance between every sample and every centroid. Then new centroid 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 horizontally (to eliminate redundant traversals over the same domain). 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 example 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 implementation, we use the Delite compiler framework [46] to stage applications. For the remainder of this paper, we will assume a high-level translation layer from user code to PPL exists and simply always start from the parallel pattern representation.

4. Pattern Transformations

One of the key challenges of generating efficient custom architectures from high level languages is in coping with arbitrarily 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. Ideally, we also want to hide memory transfer latencies by overlapping communication with computation using hardware blocks which automatically prefetch data. 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 an intermediate representation of a program in terms of optimized parallel patterns and that well known target-agnostic transformations like fusion, code motion, struct unwrapping, and common subexpression elimination (CSE) 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 accesses 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 special generation rules to 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

High Level Language	PPL	Strip Mined PPL
// Element-wise Map **val** x: Array[Float] *// length d* x.**map**{e => 2*e}	**map**(d){i => 2*x(i)}	**multiFold**(d/b)(d)(zeros(d)){ii => xTile = x.**copy**(b + ii) (i, **map**(b)(b){i => 2*xTile(i) }) }(_)
// Sums along matrix rows **val** x: Array[Array[Float]] *// m x n* x.**map**{ row => row.**fold**(0){ (a,b) => a + b } }	**multiFold**(m,n)(m)(zeros(m)){ (i,j) => (i, acc => acc + x(i,j)) }{(a,b) => **map**(n){(j) => a(j) + b(j)} }	**multiFold**(m/b0,n/b1)(m)(zeros(m)){ (ii,jj) => xTile = x.**copy**(b0 + ii, b1 + jj) tile = **multiFold**(b0,b1)(b0)(zeros(b0)){ (i,j) => (i, acc => acc + xTile(i,j)) }{(a,b) => **map**(b0){i => a(i) + b(i)} } (ii, acc => **map**(b0){j => acc(j) + tile(j)}) }{(a,b) => **multiFold**(m/b0)(m)(zeros(m)){ii => aTile = a.**copy**(b0 + ii) bTile = a.**copy**(b0 + ii) (ii, acc => **map**(b0){i => aTile(i) + bTile(i)}) }{(a,b) => **map**(m){i => a(i) + b(i)}} }
// Simple Filter val x: Array[Float] *// length d* x.**flatMap**{ e => if (e > 0) e else [] }	**flatMap**(d){i => if (x(i) > 0) x(i) else [] }	**flatMap**(d/b)(1){ii => eTile = x.**copy**(b + ii) **flatMap**(b){i => if (eTile(i) > 0) eTile(i) else [] }}
// Histogram Calculation val x: Array[Float] *// length d* x.**groupByFold**(0){ r => (r/10, 1) }{ (a,b) => a + b }	**groupByFold**(d)(0){i => (x(i)/10, 1) }{(a,b) => a + b }	**groupByFold**(d/b)(0){ii => xTile = x.**copy**(b + ii) **groupByFold**(b)(0){i => (xTile(i)/10, 1) }{(a,b) => a + b} }{(a,b) => a + b}

Table 2. Examples of the parallel pattern strip mining transformation on Map, MultiFold, FlatMap, and GroupByFold

High Level Language	Strip Mined PPL	Interchanged PPL
// Matrix Multiplication x: Array[Array[Float]] *// m x p* y: Array[Array[Float]] *// p x n* z = x.**map**{row => y.**map**{col => row.zipWith(col){(a,b) => a * b }.**sum** } }	**multiFold**(m/b0,n/b1)(m,n)(zeros(m,n)){(ii,jj) => ((ii,jj), zTile => **map**(b0,b1){(i,j) => tile = **multiFold**(p/b2)(1)(0){ kk => xTile = x.**copy**(b0 + ii, b2 + kk) yTile = y.**copy**(b2 + kk, b1 + jj) dprod = **fold**(b2)(0){ k => acc => acc + xTile(i,k) * yTile(k,j) }{(a,b) => a + b}) (0, elemTile => elemTile + dprod) }{(a,b) => a + b} zTile(i,j) + tile }) }(_)	**multiFold**(m/b0,n/b1)(m,n)(zeros(m,n)){(ii,jj) => tile = **multiFold**(p/b2)(b0,b1)(...){kk => xTile = x.**copy**(b0 + ii, b2 + kk) yTile = y.**copy**(b2 + kk, b1 + jj) (0, elemTile => **map**(b0,b1){(i,j) => dprod = **fold**(b2)(0){ k => acc => acc + xTile(i,j) * yTile(j,k) }{(a,b) => a + b} elemTile(i,j) + dprod }) }{(a,b) => **map**(b0,b1){(i,j) => a(i,j) + b(i,j) } ((ii,jj), zTile => **map**(b0,b1){(i,j) => zTile(i,j) + tile(i,j)} }) }(_)

Table 3. Example of the pattern interchange transformation applied to matrix multiplication.

examples, using similar syntax as array *slice*. We assume in these examples that 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 val-

ues 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 interchange the perfectly nested strided MultiFold and the unstrided 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 controller*s, 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

```
1  (sums,counts) = multiFold(n/b0)((k,d),k)(...){ii =>
2    pt1Tile = points.copy(b0 + ii, *)
3    multiFold(b0)((k,d),k)(zeros(1,d),0){ i =>
4      pt1 = pt1Tile.slice(i, *)
5      minDistWithIndex = multiFold(k/b1)(1)((max, -1)){ jj =>
6        pt2Tile = centroids.copy(b1 + jj, *)
7        minIndTile = fold(b1)((max,-1)){ j =>
8          pt2 = pt2Tile.slice(j, *)
9          dist = distance(pt1, pt2)
10         acc => if (acc._1 < dist) acc else (dist, j+jj)
11       }{ (a,b) => if (a._1 < b._1) a else b }
12
13       (0, acc =>
14         if (acc._1 < minIndTile._1) acc else minIndTile)
15     }{(a,b) =>
16       if (a._1 < b._1) a else b
17     }
18
19
20     minDistIndex = minDistWithIndex._2
21     sumFunc = ... // Fig 4: lines 17-20
22     countFunc = ... // Fig 4: line 21
23     (sumFunc, countFunc)
24   }{(a,b) => ... /* Tiled combination function */ }
25   (0, acc => ... /* Tiled combination function */ )
26 }{(a,b) => ... /* Tiled combination function */ }
27
28 newCentroids = multiFold(k/b1,d)(k,d)(...){ (ii,jj) =>
29   sumsBlk = sums.copy(b1 + ii, *)
30   countsBlk = counts.copy(b1 + ii)
31   (ii, acc => map(k,d){ (i,j) =>
32     sumsBlk(i,j) / countsBlk(i)
33   })
34 }
```

(a) Strip mined k-means in PPL.

```
1  (sums,counts) = multiFold(n/b0)((k,d),k)(...){ ii =>
2    pt1Tile = points.copy(b0 + ii, *)
3    minDistWithInds = multiFold(k/b1)(b1)(map(b1)((max, -1))){ jj =>
4      pt2Tile = centroids.copy(b1 + jj, *)
5      minIndsTile = map(b0){ i =>
6        pt1 = pt1Tile.slice(i, *)
7        minIndTile = fold(b1)((max,-1)){ j =>
8          pt2 = pt2Tile.slice(j, *)
9          dist = distance(pt1, pt2)
10         acc => if (acc._1 < dist) acc else (dist, j+jj)
11       }{ (a,b) => if (a._1 < b._1) a else b }
12     }
13     (0, acc => map(b0){ i =>
14       if (acc(i)._1 < minIndsTile(i)._1) acc else minIndsTile(i) })
15   }{(a,b) =>
16     map(b0){i => if (a(i)._1 < b(i)._1) a(i) else b(i) }
17   }
18   multiFold(b0)(k,d)(zeros(k,d)){ i =>
19     pt1 = pt1Tile.slice(i, *)
20     minDistIndex = minDistWithInds(i)._2
21     sumFunc = ... // Fig 4: lines 17-20
22     countFunc = ... // Fig 4: line 21
23     (sumFunc, countFunc)
24   }{(a,b) => ... /* Tiled combination function */ }
25   (0, acc => ... /* Tiled combination function */ )
26 }{(a,b) => ... /* Tiled combination function */ }
27
28 newCentroids = multiFold(k/b1,d)(k,d)(...){ (ii,jj) =>
29   sumsBlk = sums.copy(b1 + ii, *)
30   countsBlk = counts.copy(b1 + ii)
31   (ii, acc => map(k,d){ (i,j) =>
32     sumsBlk(i,j) / countsBlk(i)
33   })
34 }
```

(b) Pattern Interchanged k-means in PPL.

```
1     For each tile of b0 points:
2       Copy the points tile into local memory
3 - 4   For each point pt1 in the points tile:
5         For each tile of b1 centroids:
6           Copy the centroids tile into local memory
7 - 8       For each centroid pt2 in the centroids tile:
9             Compute distance between pt1 and pt2
10-11         Keep the closest (index,distance) pair
            End
13-16       Keep the closest pair across tiles
          End
20        Extract the index of the closest centroid
21        Add pt1 to row minDistIndex
22        Increment count at minDistIndex
24        Add point and count sums across tiles
        End
25-26   Add point and count sums across tiles
      End

28    For each tile of b1 point sums and counts:
29      Copy the point sums tile into local memory
30      Copy the point counts tile into local memory
31-32   Compute each new centroid as sums(i) / count(i)
      End
```

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

```
1     For each tile of b0 points:
2       Copy the points tile into local memory
3       For each tile of b1 centroids:
4         Copy the centroids tile into local memory
5 - 6     For each point pt1 in the points tile:
7 - 8       For each centroid pt2 in the centroids tile:
9             Compute distance between pt1 and pt2
10-11         Keep the closest (index,distance) pair
            End
          End
13-16     For each point: keep the closest pair across tiles
        End
18-19   For each point pt1 in points tile:
20        Extract the index of the closest centroid
21        Add pt1 to row minDistIndex
22        Increment count at minDistIndex
24        Add point and count sums across tiles
        End
25-26   Add point and count sums across tiles
      End

28    For each tile of b1 point sums and counts:
29      Copy the point sums tile into local memory
30      Copy the point counts tile into local memory
31-32   Compute each new centroid as sum / count
      End
```

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

	Fused		Strip Mined		Interchanged	
	Main Memory Reads	On-Chip Storage	Main Memory Reads	On-Chip Storage	Main Memory Reads	On-Chip Storage
points	$n \times d$	d	$n \times d$	$b_0 \times d$	$n \times d$	$b_0 \times d$
centroids	$n \times k \times d$	d	$n \times k \times d$	$b_1 \times d$	$(n/b_0) \times k \times d$	$b_1 \times d$
minDistWithIndex	0	2	0	2	0	$2 \times b_0$

(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.

Template		Description	IR Construct
Memories	Buffer	On-chip scratchpad memory	Statically sized array
	Double buffer	Buffer coupling two stages in a metapipeline	Same as metapipeline controller
	Cache	Tagged memory to exploit locality in random memory access patterns	Non-affine accesses
Pipelined Execution Units	Vector	SIMD parallelism	Map over scalars
	Reduction tree	Parallel reduction of associative operations	MultiFold over scalars
	Parallel FIFO	Used to buffer ordered outputs of dynamic size	FlatMap over scalars
	CAM	Fully associative key-value store	GroupByFold over scalars
Controllers	Sequential	Controller which coordinates sequential execution	Sequential IR node
	Parallel	Task parallel controller. Simultaneously starts all member modules when enabled, signals done when all members finish	Independent IR nodes
	Metapipeline	Controller which coordinates execution of nested parallel patterns in a pipelined fashion	Outer parallel pattern with multiple inner patterns
	Tile memory	Memory command generator to fetch tiles of data from off-chip memory	Transformer-inserted array copy

Table 4. Hardware templates used in hardware code generation.

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 accesses 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

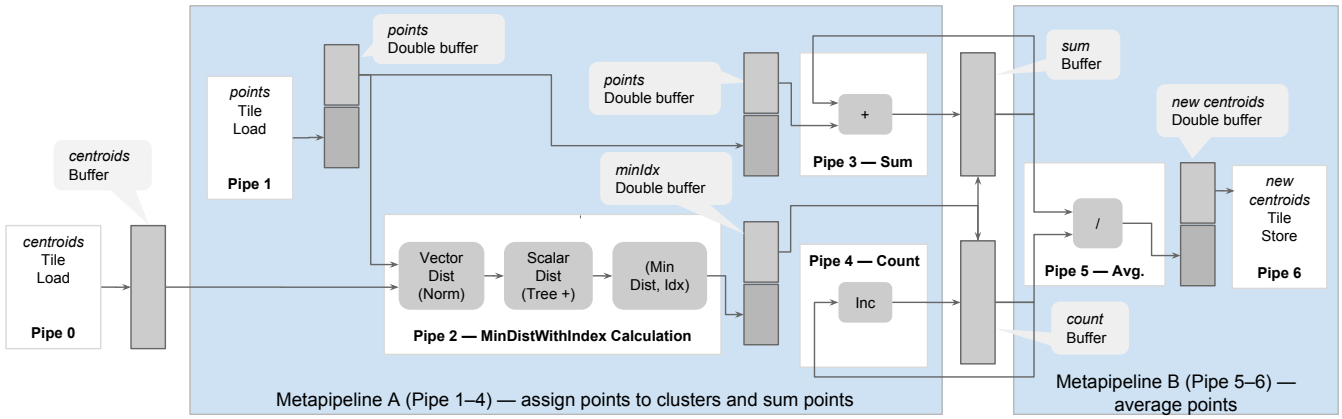

Figure 6. Hardware generated for the k-means application.

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 optimized design described by Hussain et al. [27]. While the manual implementation assumes a fixed number of clusters and a small input dataset which can be preloaded onto the FPGA, we use tiling to automatically generate buffers and tile load units to handle arbitrarily sized data. Like the manual implementation, we automatically parallelize across centroids and vectorize the point distance calculations. As we see from the k-means example, our approach enables us to automatically generate high quality hardware implementations which are comparable to manual designs.

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.

Benchmark	Description	Collections Ops
outerprod	Vector outer product	*map*
sumrows	Summation through matrix rows	*map, reduce*
gemm	Matrix multiplication	*map, reduce*
tpchq6	TPC-H Query 6	*filter, reduce*
logreg	Logistic regression	*map, reduce*
gda	Gaussian discriminant analysis	*map, filter, reduce*
blackscholes	Black-Scholes option pricing	*map*
kmeans	k-means clustering	*map, groupBy, reduce*

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

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

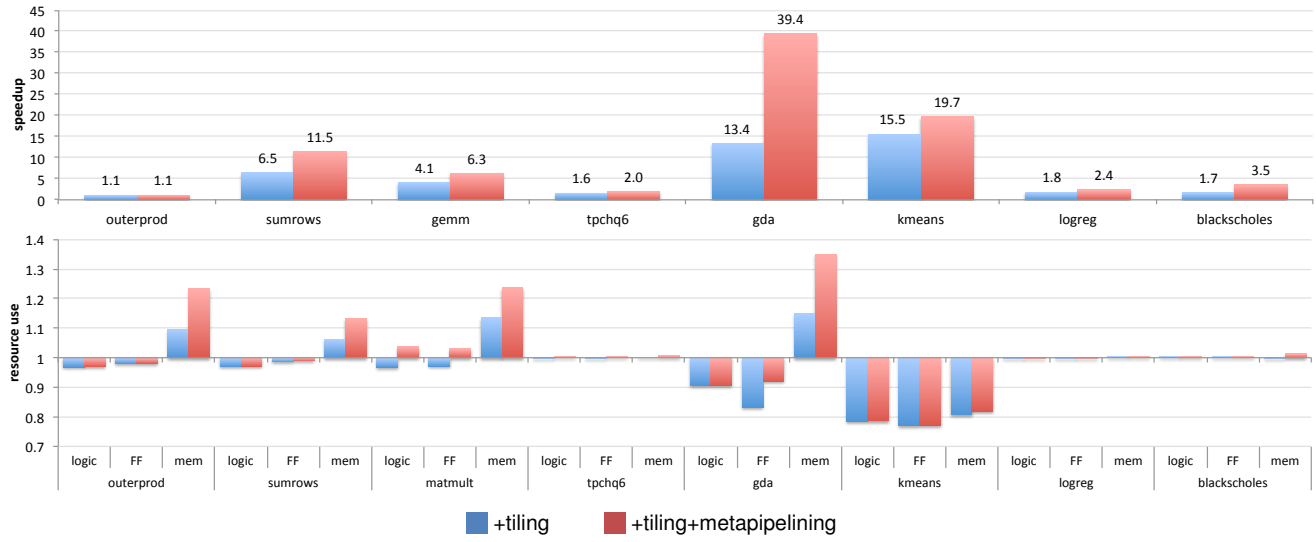

Figure 7. Speedups and resource usages, relative to respective baseline designs, resulting from tiling and metapipelining.

times to account for small runtime variations in main memory accesses and Maxeler's device driver stack.

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 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 utilizations 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 $4\times$ 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 $1.8x$ 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 $13.4\times$ and $15.5\times$ 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 *tpchq6* 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 metapipelines 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 AHPCRC 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

[1] Vivado high-level synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2016.

[2] Sadaf R Alam, Pratul K Agarwal, Melissa C Smith, Jeffrey S Vetter, and David Caliga. Using fpga devices to accelerate biomolecular simulations. *Computer*, (3):66–73, 2007.

[3] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 249–, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[5] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[6] David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[7] Donald G Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011.

[8] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303, Paphos, Cyprus, March 2010. Springer Verlag.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[11] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *International Symposium on Code Generation and Optimization,*, CGO, 2016.

[12] Samuel Brown et al. Performance comparison of finite-difference modeling on cell, fpga and multi-core computers. In *SEG/San Antonio Annual Meeting*, 2007.

[13] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

[14] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI. ACM, 2010.

[15] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[16] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[17] Christian de Schryver. *FPGA Based Accelerators for Financial Applications*. Springer, 2015.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, OSDI, pages 137–150, 2004.

[19] S.A. Edwards. The challenges of synthesizing hardware from c-like languages. *Design Test of Computers, IEEE*, 23(5):375–386, May 2006.

[20] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[21] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. PollyâĂŤperforming polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[22] Frederik Grull and Udo Kebschull. Biomedical image processing and reconstruction with dataflow computing on fpgas. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–2. IEEE, 2014.

[23] Prabhat K. Gupta. Xeon+fpga platform for the data center. `http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf`, 2015.

[24] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 223–238, New York, NY, USA, 2015. ACM.

[25] Eric Hielscher. *Locality Optimization For Data Parallel Programs*. PhD thesis, New York University, 2013.

[26] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 1–15, Santa Clara, CA, July 2015. USENIX Association.

[27] H.M. Hussain, K. Benkrid, H. Seker, and A.T. Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255, June 2011.

[28] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Micro, 2014.

[29] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. Cgpa: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 78:1–78:6, New York, NY, USA, 2014. ACM.

[30] Maxeler Technologies. MaxCompiler white paper, 2011.

[31] Oskar Mencer, Erik Vynckier, James Spooner, Stephen Girdlestone, and Oliver Charlesworth. Finding the right level of abstraction for minimizing operational expenditure. In *Proceedings of the fourth workshop on High performance computational finance*, pages 13–18. ACM, 2011.

[32] M. Odersky. Scala. `http://www.scala-lang.org`, 2011.

[33] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for largescale dnn systems. Hot Chips 26, 2014.

[34] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware. Technical report, Microsoft Research, February 2015.

[35] D. Petkov, R. Harr, and S. Amarasinghe. Efficient pipelining of nested loops: unroll-and-squash. In *Parallel and Dis-*

tributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, pages 6 pp–, April 2002.

[36] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, feb 1999.

[37] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[38] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, January 2011. ACM Press.

[39] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[40] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[41] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. Chimps: A high-level compilation flow for hybrid cpu-fpga architectures. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 261–261, New York, NY, USA, 2008. ACM.

[42] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[43] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[44] Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society.

[45] M.C. Smith, Jeffrey S Vetter, and Sadaf R. Alam. Scientific computing beyond CPUs: FPGA implementations of common scientific kernels. In *Proceedings of the 8th Annual Military and Aerospace Programmable Logic Devices International Conference*, 2005.

[46] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. In *TECS'14: ACM Transactions on Embedded Computing Systems*, July 2014.

[47] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[48] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

[49] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 78–85, Piscataway, NJ, USA, 2015. IEEE Press.

[50] The Khronos Group. OpenCL 2.0. `http://www.khronos.org/opencl/`.

[51] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, New York, NY, USA, 2015. ACM.

[52] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[53] GL Zhang, Philip Heng Wai Leong, Chun Hok Ho, Kuen Hung Tsoi, Chris CC Cheung, Dong-U Lee, Ray CC Cheung, and Wayne Luk. Reconfigurable acceleration for monte carlo based financial simulation. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 215–222. IEEE, 2005.

[54] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.

[55] Ling Zhuo and Viktor K Prasanna. High-performance designs for linear algebra operations on reconfigurable hardware. *Computers, IEEE Transactions on*, 57(8):1057–1071, 2008.