Efficient Memory Partitioning in Software Defined Hardware

Matthew Feldman
Electrical Engineering
Stanford University
United States
mattfel@stanford.edu

Tian Zhao
Electrical Engineering
Stanford University
United States
tianzhao@stanford.edu

Kunle Olukotun
Electrical Engineering and Computer Science
Stanford University
United States
kunle@stanford.edu

Abstract
As programmers turn to software-defined hardware (SDH) to maintain a high level of productivity while programming hardware to run complex algorithms, heavy-lifting must be done by the compiler to automatically partition on-chip arrays. In this paper, we introduce an automatic memory partitioning system that can quickly compute more efficient partitioning schemes than prior systems. Our system employs a variety of resource-saving optimizations and an ML cost model to select the best partitioning scheme from an array of candidates. We compared our system against various state-of-the-art SDH compilers and FPGAs on a variety of benchmarks and found that our system generates solutions that, on average, consume 40.3% fewer logic resources, 78.3% fewer FFs, 54.9% fewer Block RAMs (BRAMs), and 100% fewer DSPs.

ACM Reference Format:
Matthew Feldman, Tian Zhao, and Kunle Olukotun. 2022. Efficient Memory Partitioning in Software Defined Hardware. In Proceedings of Work in Progress (arXiv Preprint). ACM, New York, NY, USA, 12 pages.

1 Introduction
In recent years, there has been a growing demand for compute devices that can efficiently run increasingly complex algorithms. Experts from a wide variety of domains, such as Machine Learning [4], Computational Physics [8], and Genomics [39], are facing unprecedented challenges in processing massive data sets quickly and efficiently. Spatial architectures, such as field-programmable gate arrays (FPGAs) and reconfigurable dataflow architectures (RDAs), provide performance-per-watt advantages over CPUs and GPUs [20] and are becoming popular for these kinds of domain experts. They allow the programmer to create a digital circuit that can act as an accelerator for a particular algorithm. Hardware accelerators perform computation by flowing data through a pipelined circuit, rather than by serially executing a sequence of instructions. Unfortunately, they traditionally require complicated register-transfer level (RTL) languages, like VHDL and Verilog, that domain experts often find prohibitively difficult to use.

For this reason, software-defined hardware (SDH) has emerged as an active research field to solve this problem of programmability without sacrificing performance. For certain domains, there are domain specific languages (DSLs), such as Halide [28], DNNWeaver [36], Aetherling [15], RIPL [38], and SODA [41], which limit what the programmer can express in order to generate high performance designs for a particular domain. There are also numerous general purpose SDH languages available today, including Vivado High Level Synthesis (HLS) [1], SDAccel [3], OpenCL [2], and Spatial [23]. These languages are embedded in high level software languages, such as C or Scala, and allow the programmer to use high level abstractions to create arbitrary hardware accelerators.

SDH languages allow the programmer to exploit both pipeline and spatial parallelism by nesting loops and annotating iterators. Pipeline parallelism refers to concurrent execution of consecutive stages in a computation graph, and spatial parallelism refers to the concurrent execution of the same stage in a computation graph. These frameworks also allow the programmer to declare multidimensional arrays, or memories, that reside on-chip. The interplay between parallelism and how these memories are accessed in general-purpose SDH languages is the focus of this paper. "Memory banking" is the process in which the compiler decides how to manage concurrent accesses to on-chip memories by partitioning them across multiple physical resources.

In the worst case, a banking system that is not well optimized could increase the compile time of an SDH program by minutes to hours, as we show in this paper. This increase in the programmer’s development cycle could significantly hinder the productivity of the SDH tool. Furthermore, a poorly optimized banking system could also result in wildly inefficient banking schemes. A naive banking scheme could result in more resources being dedicated to the implementation of the banking scheme than those dedicated to the datapath of the algorithm that does actual computational work. These two issues could render an SDH framework entirely unusable for a performance-oriented programmer.
State-of-the-art implementations of banking analysis were generally designed for certain kinds of compute patterns in mind, such as linear algebra [23] and stencil operations [42]. However, these approaches do not scale when presented with more challenging banking problems that are found in applications with more dynamic access patterns, larger parallelization factors, and more complex control structures.

In this paper, we introduce a novel banking system capable of automatically solving challenging banking problems with solutions that are more efficient than existing state-of-the-art tools. Our system makes the following contributions:

- Use heuristics to quickly identify a collection of valid banking schemes (Section 3.3).
- Apply targeted transformations to the datapath associated with each solution. (Section 3.4).
- Rapidly estimate resource utilization for each solution using a machine learning (ML) pipeline (Section 3.5).

Figure 1 shows the conceptual view of our system. The input is logical accesses to an array, coupled with information about the concurrency of these accesses. The output is an elaborated, retimed circuit that implements memory virtualization to satisfy the constraints of the original program.

We compared our system against various state-of-the-art SDH compilers on a variety of benchmarks and found that our system generates solutions that, on average, consume 40.3% fewer logic resources, 78.3% fewer FFs, 54.9% fewer BRAMs, and 100% fewer DSPs without losing performance.

2 Background

Memory banking is the process of partitioning a multidimensional array so that it can be implemented in hardware as a collection of physical BRAM resources and surrounding logic to index into them. In this paper, we refer to the access pattern to an array in the program as the logical access (i.e. arr[2*1+3]), and the circuits which connect to BRAMs as physical accesses.

2.1 Representing The Problem Using the Polyhedral Model

The polyhedral model is a formalism for representing a program’s iteration space, memory space, and access patterns algebraically for a compiler to statically analyze [21]. It has been used by others to solve various problems such as synthesis, verification, and optimization of systolic arrays, detecting parallelism and efficiently scheduling programs in parallel compute environments, and on-chip memory partitioning in dataflow architectures [5, 16–18, 29–33, 40, 42].

In this work, we use it as a tool to solve the banking problem, and briefly summarize the important concepts used in prior work as it relates to memory partitioning on FPGAs.

Definition 2.1. A Polyhedron is a set of points in n-dimensional space that satisfy a set of linear inequalities. Namely, the set of points \( P = \{ x \in \mathbb{Q}^n | a \cdot x \leq b \} \) where \( \mathbb{Q}^n \) is an n-dimensional vector of rational numbers, \( A \) is a matrix where \( A_{i,j} \in \mathbb{Q} \), and \( b \in \mathbb{Q}^n \).

Definition 2.2. A Polytope is a bounded polyhedron, e.g. one that contains a finite amount of integer points.

Definition 2.3. A Parallelotope is a polytope where every pair of opposite facets are congruent and parallel, i.e. the n-dimensional generalization of a parallelogram (2D) or parallelepiped (3D).

Definition 2.4. The iterator space is a polytope defined by the iterators, \( i \), of a loop nest.

Definition 2.5. An array is an n-dimensional on-chip memory whose addresses represent a polytope.

Definition 2.6. An access pattern for access \( a \) is an expression representing the mapping from the iterator space to an n-dimensional reference to the array polytope.

Definition 2.7. An access group is a collection of access patterns which can be active during the same cycle at runtime.
Efficient Memory Partitioning in Software Defined Hardware

| Type | Name | Description |
|------|------|-------------|
| Input | $\tilde{D}$ | Dimensions of memory, $|\tilde{D}| = n$ |
|       | $\tilde{a}$ | Logical accesses, grouped by concurrency |
|       | $\tilde{i}$ | Iterators in the scope of an access |
|       | $\tilde{x}$ | Memory address reference for an access, $|\tilde{x}| = n$ |
|       | $k$ | Number of ports on underlying BRAM |
| Solve | $\tilde{N}$ | Number of banks, $|\tilde{N}| \in \{1, n\}$ |
|       | $\tilde{B}$ | Blocking factor, $|\tilde{B}| = |\tilde{N}|$ |
|       | $\tilde{a}$ | Partition vector, $|\tilde{a}| = n$ |
|       | $\tilde{P}$ | Partition parallelopetope, $|\tilde{P}| = n$ |
| Metrics | $FO_a$ | # of banks an access touches (Fan-Out), $FO_a \leq \prod N$ |
|       | $FI_b$ | # of accesses a bank feeds (Fan-In), $\sum FI_b = \sum FO_a$ |
|       | $\delta$ | Per-dimension padding |

Table 1. Partitioning parameters and definitions.

sketched in Figure 2. Under hyperplane partitioning, the array polytope is divided into parallel “hyperplanes,” where each hyperplane represents a bank. Under lattice partitioning, the array is divided into a tessellation of congruent parallelopetopes, where each position within the parallelopetope is assigned to a unique bank. While most common partitioning problems have both hyperplane and lattice solutions, there are certain patterns where only one can provide a solution. For example, hyperplane partitioning can solve problems that require a block-cyclic pattern of banks, while lattice partitioning can solve problems with concurrent accesses that do not lie within a rectangular parallelopetope.

In this paper, we introduce a system based on hyperplane partitioning with an extension that captures a subset of lattice partitioning solutions, namely those composed of orthogonal parallelopetopes. These are referred to as “multidimensional” hyperplane geometries and are discussed in 3.3.

Table 1 concisely summarizes all of the quantities involved in our banking system. Input parameters are provided by the SDH framework and represent the access pattern the programmer specified in the code. Solve parameters are computed by our banking system. Our system internally computes a collection of these but only returns the optimal set. Metrics which are helpful quantities for understanding how the partitioning scheme will map to hardware.

Equations 1 and 2 show how the solve parameters are used to compute a bank address (BA) and offset (BO). We refer to these collectively as the “bank resolution” equations.

$$BA = \lfloor \frac{\tilde{x} \cdot \tilde{a}}{B} \rfloor \mod N$$  \hspace{1cm} (1)

$$BO = \left[ B \cdot \sum_{i=0}^{n} \left( \frac{\tilde{x}_i}{P_j} \right) \right] + (\tilde{x} \cdot \tilde{a} \mod B)$$ \hspace{1cm} (2)

The equation for BA divides the array polytope into parallel hyperplanes, and we therefore refer to $\tilde{a}$, $N$, and $B$ as representing a “hyperplane geometry.” The parameter $\tilde{P}$ is only used to compute a physical offset, and a hyperplane geometry may have many valid choices for $\tilde{P}$. It is related to the periodicity of Equation 1 and represents a region in the array such that each BA appears at least once and no more than $B$ times.

**Definition 2.8.** A conflict polytope is the polytope generated by applying Equation 1 to the delta between the address patterns of two different accesses $(BA(\tilde{x}_{a1} - \tilde{x}_{a2}))$.

**Definition 2.9.** A hyperplane geometry is deemed valid for a $k$-ported memory if no set of $k$ accesses contain more than $k – 1$ pairs of non-empty conflict polytopes.

These definitions allow our system to handle addresses with non-affine components using quantifier-free Presburger arithmetic [37]. Namely, any function call without side-effects (including indirection arrays) that is used in an address expression (i.e. $f(\tilde{b}_0)$ in arr($f(\tilde{b}_0) + i_1$)) can be represented an uninterpreted function symbol. This means that even though the compiler cannot analyze $BA(f(\tilde{b}_0) + i_1)$, it can analyze any conflict polytope it generates against another access containing the same function symbol.

Finally, $FO_a$ and $FI_b$ describe the fan-out of accesses and fan-in of banks, respectively. They describe the size of the crossbars required to arbitrate between accesses and banks. If $\tilde{P}_i$ does not evenly divide $\tilde{D}_i$, then the banking equations result in mathematically “inaccessible” elements that require the compiler to pad the array with $\tilde{B}$. When $B > 1$, there may be more inaccessible elements due to unequal representation of each bank within a $\tilde{P}$ region.

### 2.3 Consequences of Parameters

The values of $\tilde{N}$, $\tilde{B}$, $\tilde{a}$, and $\tilde{P}$ have implications on the overall FPGA resource utilization and latency of the circuit, since they impact the bank resolution logic, crossbar sizes, and bank volumes which must all map to quantized resources on the FPGA.

Consider the snippet in Figure 3a with access patterns $6 \cdot i + 1, 6 \cdot i + 2, 6 \cdot i + 4,$ and $6 \cdot i + 5$ (we substitute $k$ for unit-step iterator $i$). Figure 3b shows three potential ways this problem can be solved.

Option 1 uses 5 banks and full crossbars for each access ($FO_a = 5$). Option 2 appears better since it uses only 4 banks and $FO_a = 1$, but requires a costly divide-by-3 and a multiply-by-2 operation. Option 3 uses 6 banks but achieves $FO_a = 1$ while also eliminating B and $\alpha$ arithmetic. Depending on the size of the array and the target FPGA, the volume of one
\[ m[k+1] + m[k+2]; \]

for \( k = 0 \rightarrow 3 \rightarrow M \) \text{ par} 2

\[ = m[k+1] + m[k+2]; \]

Figure 3. (a) Sample access pattern with iterator \( k \) start=0, step=3, stop=M, and parallelization=2. (b) Three valid banking schemes for \( m \) with \( BA \) values shown.

"bank" may spill into 2 BRAMs, so solutions with smaller \( N \) may consume more BRAMs.

The key point is that there are always many solutions to a banking problem and it is difficult to predict which one is the best. Additionally, the definition of "best" may change depending on which resource is scarcest for a given application.

### 2.4 Concepts in Hierarchically-Nested State Machine Programming

A programming model based on parallel patterns [7] is a good starting point for exposing a high level of abstraction without losing handles on performance [23]. For this reason, we chose to use Spatial as the host compiler for our banking system since it is an open-source, extensible language that can express massive design spaces, an explicit memory hierarchy, and multi-level parallelism.

Our system targets programs composed as a set of hierarchically nested state machines, or controllers. A controller is expressed as a traditional software loop and is represented in the IR as a multi-level counter chain that feeds iterator values to a basic block. A multi-level counter chain is a counter that spans a multidimensional iteration space whose bounds do not need to be compile-time static.

#### 2.4.1 Controller Level

The contents of a controller’s basic block define it’s level:

- **Inner controllers** only contain dataflow graphs made up of primitives.
- **Outer controllers** only contain other controllers

An outer controller and the controllers in its basic block define a parent-child relationship in the controller hierarchy. An outer controller’s width is the number of children it contains. A controller’s sub-tree is the set of controllers found by following its children recursively. A node’s ancestors is a list of every controller in the hierarchy that encloses it. The least common ancestor (LCA) of two nodes is the most deeply-nested controller which they both share.

#### 2.4.2 Controller Schedule

"Scheduling" refers to how nodes in a controller’s basic block execute relative to each other but has different meanings for inner and outer controllers.

For outer controllers, there are five schedules describing how the children execute in hardware:

- **Sequential** - Child controllers execute one at a time with no overlap.
- **Pipelined** - Child controllers execute in pipelined (i.e. overlapping) fashion.
- **Fork-Join** - Child controllers execute simultaneously and independently of each other.
- **Fork** - One child controller executes per iteration, based on a set of if/then/else conditions.
- **Streaming** - Child controllers execute as long as their input data is available.

For inner controllers, the schedule refers to the mapping from each node in the pipelined dataflow graph to the cycle it will execute. Scheduling nodes to execute during different cycles allows the design to safely run at a higher clock rate. Their runtime is entirely defined by their latency, which is the cycle at which its latest node is scheduled, and its initiation interval, which is the number of cycles the controller must wait before it can increment the multi-level counter and issue another iteration. The hardware-complexity of the datapath determines the latency, and the loop-carry dependencies determine the initiation interval.

#### 2.4.3 Controller Parallelization

Parallelization is one of the ways that programs mapped to FPGA can improve performance. Specifically, parallelizing a loop by \( P \) means that \( P \) consecutive iterations of the loop will be executed simultaneously. To a first-order approximation, this means that parallelization results in a factor of \( P \) performance improvement by using \( P \)-times as many resources compared to the un-parallelized loop.

In reality, this typically improves performance by less than \( P \) and increases resource utilization by more than \( P \). This is because initiation interval and latency may change with parallelization. We must distinguish between parallelization of inner and outer controllers separately to understand why.

We walk through the example snippet in Figure 4 to demonstrate this.

Parallelization applied to an inner controller results in vectorization of the datapath. Figure 5 shows how this is done for loop \( k \) with \( IP = 2 \).

Parallelization applied to an outer controller results in unrolling. Each child controller is cloned in whole and added to the hierarchy. This is shown in Figure 6 as the "Pre-Unrolled" tree structure is transformed into the "Intermediate" structure for \( OP = 2 \). The compiler must then inject Fork-Join controllers to this intermediate structure to achieve the parallelism specified in the program. There are two strategies that
Efficient Memory Partitioning in Software Defined Hardware

Figure 4. Example construction of an outer controller (Loop $i$) with parallelization ($OP$) and two child controllers (Loops $j$ and $k$). Loop $k$ has parallelization factor $IP$.

Figure 5. Dataflow path for the inner controller of loop $k$ in Figure 4 before and after applying vectorization ($IP=2$).

the compiler may employ to capture this level of parallelism, shown in the "Post-Unrolled" trees in Figure 6:

- **ForkJoin-of-Pipelines unrolling** is when the Fork-Join controllers are injected between the outer controller and child stages such that all lanes of each child are synchronized (i.e. always begin their execution simultaneously). This guarantees that child $c$ will not begin executing iteration $i+P$ until iterations $i/P*P$ to $(i/P*P) + P - 1$ are completed, for any child and iteration of the parent controller.

- **Pipeline-of-ForkJoins unrolling** is when each lane of the original outer is separated into its own controller that is structurally identical to the pre-unrolled loop. A single Fork-Join controller is injected above the outer controllers, such that all lanes begin executing simultaneously. This guarantees that child $c$ will not begin executing iteration $i+P$ until iteration $i$ is completed, for any child and iteration of the parent controller. However, there is no guarantee about the relative ordering between iteration $i/P*P$ and $i/P*P*K$, where $K \% P != 0$.

For any node in the IR, its "unroll ID" (UID) is a list of integers describing which lane of each ancestor controller it belongs to. The UID of a node is described as the "base UID" if all integers in the UID are 0.

3 Banking System Design

In this section, we begin by discussing the analysis that analyzes accesses in a program and converts them into a polytope emptiness problem. Then, we describe the three steps in our system for solving this problem:

- Compute a list of solutions (constants for $\tilde{N}$, $\tilde{B}$, $\tilde{a}$, and $\tilde{P}$) that satisfy the constraints
- Apply resource-saving optimizations to the bank resolution datapaths
- Estimate the cost of each solution and return the best

3.1 A Running Example

In order to drive the components of our banking system, we introduce a motivating example that presents an interesting banking problem. Figure 7 shows part of the grid-variant of the Molecular Dynamics (MD) benchmark from Machsuite, as it relates to the accesses to one memory.

At a high level, this algorithm models a 3-dimensional field of molecules. The molecules are grouped into "cells," which means the field is represented by the four-dimensional structure $dvec_sram$. The first three dimensions describe the location of a cell in space, and the fourth dimension enumerates each molecule within the cell. The algorithm computes how each molecule interacts with each molecule in its 26 adjacent cells.

In this snippet, we are initializing the memory in Line 1 by fetching data from DRAM to it and writing $PL$ elements per cycle in its leading dimension. Later, in Line 7, we read this data from the memory to drive the algorithm. Note the innermost controller’s counter spans $Q_{RNG}(x, y, z)$, which refers to the amount of molecules in the cell at $x$, $y$, and $z$. Each cell contains a different number of molecules, so this is a data-dependent value. There are a total of $PL$ writers and $PX*PY*PZ*PP*PQ$ readers.

3.2 Distilling the Program to a Polytope Emptiness Problem

There are three steps involved in constructing the polytope emptiness problem: group placement, address pattern extraction, and synchronization analysis.
Applying the algorithm to our running example yields a banking problem with two groups:

\[
G = \{(w_0)\}
\]

The subscript integers refer to the UID of each access. \(w_n\) refers to vectorization from \(PL\) and \(r_{ijk}\) refers to the lanes of loops \(x/y/z\) (flattened), \(p\), and \(q\).

The compiler must inspect the ancestors of each access and place it into one group. The pseudo-code in Figure 8 shows how to perform the grouping. \texttt{isConcurrent} is defined for each controller based on the following:

- **If \texttt{lca} is an inner controller**, \texttt{isConcurrent} returns true if the distance (in cycles) between the scheduling of \(a\) and \(b\) is less than the initiation interval of the \texttt{lca}.
- **If \texttt{lca} is an outer controller**, \texttt{isConcurrent} returns true if the schedule is Fork-Join or Stream and returns false if the schedule is Sequential, Fork, or Pipelined scheduling. Note in the case of a Pipelined controller, the accesses are concurrent but routed to different buffers.

The \texttt{lca} of two accesses is often a Fork-Join controller due to the unrolling strategies described in Section 2.4.3. Applying the algorithm to our running example yields a banking problem with two groups:

\[
G = \{(w_0, w_1, ..., w_{PL-1}),
\{r_{000}, r_{001}, ..., r_{PX*PY*PZ-1,PP-1,PQ-1}\}\}
\]
we produce a $\mathcal{B}$A projections separately with a 1-dimensional hyperplane geometry based in these schemes, we bank each dimension of the memory [42], our system also searches for multidimensional schemes. Multidimensional Banking In addition to flat schemes in 3.4. All other values are de-prioritized. To all parameters that can be broken down by rules introduced in 3.3 Building a Solution Set Our solution set is a collection of $\bar{N}$, $\bar{B}$, $\bar{a}$, and $\bar{P}$ tuples which satisfy the banking constraints. Finding this set requires us to build candidate sets for $\bar{N}$, $\bar{B}$, and $\bar{a}$, and check for combinations that can be proven to be valid geometries for the given access pattern. Then various possible values for $\bar{P}$ are calculated for each geometry.

Prioritizing Candidate Sets In order to increase the probability of finding a “good” solution quickly, we prioritize certain parameters in these candidate sets. Specifically, we find the LCM of the access groups’ sizes and prioritize the first few multiples of this. This is more likely to find schemes that do not require full cross-bars between the accesses and the banks (e.g. small FO$_a$). We also remove candidates for $\bar{a}$ if the elements are not mutually co-prime with the element(s) in $\bar{B}$, since the same geometry can be expressed by dividing all parameters by the GCD. Finally, we prioritize integers for all parameters that can be broken down by rules introduced in 3.4. All other values are de-prioritized.

Multidimensional Banking In addition to flat schemes [42], our system also searches for multidimensional schemes. In these schemes, we bank each dimension of the memory separately with a 1-dimensional hyperplane geometry based on the projections of the original accesses. This means that we produce a BA per-dimension (BO is still a scalar, namely the intra-bank offset of a multidimensionally-indexed bank) Multidimensional schemes is the subset of lattice-indexed schemes based on orthogonal parallelopetes. Multidimensional banking schemes can also be verified more quickly. This is because $\binom{c}{k}$ polytope-emptiness checks must occur to prove $t$ concurrent accesses are properly banked for a $k$-ported memory. The complexity of verification is therefore $O(t^k)$, since $k$ is typically much smaller than $t$. Projecting accesses results in smaller groups per dimension, either due to redundancy (two projections are identical) or regrouping (two accesses are guaranteed to never conflict because BA on at least one other dimension always differs). This reduction is most significant when the accesses are heavily-parallelized on multiple dimensions, allowing the system to verify banking solutions more rapidly.

Fewer-Ported Solutions If the underlying BRAM supports $k$ ports, there may be some area overhead associated with the memory template when there are more than $k$ accesses that may connect to one bank. For this reason, our analyzer adds sub-$k$-ported solutions to the solution set.

Bank-by-duplication Our system iteratively partitions readers into separate groups and routes each group to a different duplicate of the array in certain cases. It then re-runs the banking analysis on each duplicate separately. Banking-by-duplication is occasionally the best strategy in cases where LUTs are scarce but BRAMs are abundant.

3.4 Resource-Saving Datapath Transforms The banking resolution logic (Equations 1 and 2) include multiplication, division, and modulo operations that may be costly on an FPGA. Because our banking analyzer has the freedom to choose the actual constants used in these equations, we can aim for those constants that allow for resource-saving transformations and avoid calling vendor-specific arithmetic IPs.

Crandall’s Algorithm We apply Crandall’s algorithm to perform division and modulo by Mersenne numbers (i.e. those in the form $M = 2^n - 1$). This allows us to perform a cascade of bit-wise operations and simple additions rather than calling division or modulo IPs in hardware. We further rewrite modulo operations when the operand can evenly divide a Mersenne number. Specifically, if $M_2 \cdot k = 2^n - 1$ for some $1 \leq k < R$, then we can apply Crandall’s algorithm on $M$ followed by a $k$-wide one-hot mux to compute $x \mod M_2$.

$$x \mod M_2 \equiv (x \mod M) \mod M_2 \quad (6)$$

For reference, there are 5 Mersenne integers, 5 integers that evenly divide a Mersenne integers with R=16, and 6 power-of-2 integers between 1 and 65. This provides a sufficiently large pool of desirable constants towards which we can steer our system’s search.
Partition Factor (alpha)

Number of Banks
Blocking Factor
Dimensions (dia)
Bit Precisions (prec)
Partition Factor

Template features

Number of Banks
Blockin g Factor
Dimensions (dia)
Bit Precisions (prec)
Partition Factor (alpha)

Subgraph features

−

Banking resolution path
Banked memory
Writer

Feature Score

Banking resolution path
Banked memory
Reader!

Flow

−

Subgraph features

Gradient-boosting
decision tree

Feature re-selection

Polynomial feature
generator

Figure 10. A machine learning architecture to predict a banking scheme’s resources after PnR.

### Binary decomposition of multiplication

We also optimize multiplication in the form of \( a \times c \), when \( c \) is a constant in the form \( c = \sum_{0 \leq k < R} S(k)2^{n_k} \), where \( S(k) \in \{\pm 1\} \) and \( R \) is the radius of the optimization. This allows the compiler to apply the rewrite rule \( a \times c = \sum_k a \times S(k)2^{n_k} \), which is simply the sum of bit-shifts of \( a \).

For reference, with \( R=2 \), half of the integers between 1 and 65 can be rewritten using only bit-shifts and addition.

### 3.5 Machine Learning Model for Resources Estimation

Finally, the compiler chooses a valid banking scheme requiring minimal resources, i.e., the RTL code generated from the scheme leads to the least hardware resource after the downstream FPGA toolchain completes PnR. Previous work [42] used an analytical model to estimate the hardware resource. However, we found that building an accurate analytical model is very challenging due to the complexity of our RTL templates. Instead, we built a machine learning pipeline to provide the searching process with reasonable estimates of the hardware resources required for a banking scheme.

#### 3.5.1 Architecture of Resource Estimator

Previous work [24] used a Multi-layer Perceptron (MLP) model for hardware resource estimation. From our experiments, we found that MLP performed worse on small datasets due to overfitting. Hence, we built a pipeline based on decision trees and fine-tuned it to reduce overfitting. Figure 10 shows the pipeline’s architecture. It takes two classes of features from the parameters of a banking scheme:

- **Template features** that include primitiv es and derived parameters.

- **Subgraph features** that include neighbors and accessors of a memory node in the dataflow.

The first stage in the pipeline generates second-degree polynomial combinations of raw features. This approach helps create stronger features, e.g., the product of the number of banks over all the dimensions in a high-dimensional memory node, at the expense of generating an ample feature space that can hurt the training speed. The second stage consists of a regressor based on the gradient-boosting tree [9]. The last stage re-selects the generated features based on their importance. We define importance as the frequency each generated feature appears in the trained model. In our experiment, we found that 36 generated features provided the searching process with enough accuracy.

#### 3.5.2 Training and Fine-Tuning the Estimator

We created the dataset by using Spatial’s regression benchmark suite 1. However, our approach can be applied to any SDH framework with a reasonably explicit representation of the memory template and bank resolution logic exposed in its IR. We ran PnR on all the RTL files generated by this benchmark suite to collect the resources used for every memory and arithmetic node in each application. A sample in the dataset contains a memory node’s raw features and its resources in terms of look-up tables (LUT), flip-flops (FF), and RAMs. The created dataset contains 831 samples.

Due to the small size of this dataset, we carefully fine-tuned our proposed pipeline to control overfitting. We trained two models: a baseline MLP model similar to the one proposed in [24], and the model pipeline proposed in this work. We were not able to achieve high performance by using the original baseline model; hence, we augmented its architecture and fine-tuned it to get the best possible performance on the dataset. Specifically, we performed an exhaustive grid search on the training and regularization parameters for the baseline model and chose the one with the best performance. Due to the complexity of the proposed model, we were not able to search for the best parameters exhaustively. Hence, we focused on tuning the regularization parameters to avoid overfitting. Please refer to Table 3 in the appendix for the final parameters of each model.

For cross-validation, we randomly permuted the raw dataset ten times. Every time, we split the dataset into a training set and a test set randomly with a 7-to-3 ratio. We collected the training scores and test scores for both models. Figure 11 shows learning curves for both models when predicting the LUT resource. For learning curves showing both models predicting other resources, please refer to Appendix A.

---

1Spatial regression suite: https://github.com/stanford-ppl/spatial/tree/master/test/spatial/tests
Efficient Memory Partitioning in Software Defined Hardware

Figure 11. Learning curves of the baseline model (left) and our proposed model pipeline when fine-tuned on the dataset from Spatial’s regression suite.

In our experiment, we scored both curves using $R^2$. We averaged the score curves over all the ten splits. The colored fields show the standard deviation of all the scores collected during cross-validation. Our proposed model pipeline achieves an average $R^2$ score of 0.86, which is higher than the baseline model’s average $R^2$ score of 0.60. Besides, the model shows a smaller standard deviation of its cross-validation test scores, which indicates that it suffers less from overfitting.

4 Evaluation

We implemented our banking system as a compiler pass in Spatial [23], a fully open-source SDH framework that can be easily modified and target a variety of FPGAs. The system uses the Integer Set Library [40] to perform the polytope emptiness checks. We evaluated our system on both a Xilinx Virtex 7 and Amazon EC2 F1 instance (Xilinx VU9P) for eight stencil patterns and an additional three patterns from real-world applications described here.

Smith-Waterman SW is a sequence alignment algorithm used in genomics. It contains a dynamic programming component known as Genome Alignment using Constant-memory Traceback (GACT) [39]. This essentially creates a sliding-window access pattern where a cell is updated based on the values of its north, west, and north-west neighbor. We parallelized this access pattern by 4 to expose wavefront parallelism in the algorithm.

Sparse Matrix-Vector Multiplication SPMV is a common linear algebra kernel [35]. Our version uses an edge-list representation to identify dense regions in the matrix. We parallelized this algorithm over four rows and three columns, so that each row’s strided access pattern has a “random” relative offset. This type of pattern is a good candidate for multidimensional banking because this random offset effectively disappears from projection regrouping (see 3.3).

Stochastic gradient descent SGD is a training algorithm ubiquitous in machine learning [14]. Our version uses mini-batch, which stores a matrix of input data on-chip and has two modes of access: first in column-major to compute predictions with the model, then in row-major to compute the gradients. These two modes of access can each be parallelized across rows and columns, and will never be concurrent (i.e., two access groups). Our version parallelized both access patterns so that there are 12 accesses in each group.

4.1 Comparisons on Virtex 7

We first compared our system on eight stencil patterns against [42], called “baseline”, as well as unmodified Spatial. Unmodified Spatial uses the first valid scheme it finds. These results show that solving for numerous solutions and applying resource-saving transformations to each reveals more efficient partitioning schemes in all cases. Our system always finds parameters that result in DSP-free circuits.

4.2 Comparisons on AWS F1 (VU9P)

We also tested our system on a larger FPGA against a state-of-the-art commercial SDH framework called Merlin [45] on Amazon’s EC2 F1 instances. To the best of our knowledge, Merlin does not target the Virtex 7 FPGA so we chose to use the popular F1 backend. The Merlin and Spatial compiler often land on a partitioning solution that over-utilizes resources. The key is that our system can view a more diverse set of solutions, take advantage of the constants in the bank resolution arithmetic. The ML model is what allows our system to choose which one of the many valid solutions will be the best.

For example, the Merlin compiler appears to bank the denoise and bicubic kernels as sobel-like patterns rather than 4-point accesses, hence producing a scheme requiring 9 banks and resource-intensive arithmetic. The Spatial compiler detects the “leaner” solution for these 4-point access patterns by having $B \neq 1$. Our system recognizes both of these kinds of solutions, and improves particularly on Spatial’s base solution by applying resource-saving transformations on a true dual-ported scheme.

5 Related Work

The theory of the partitioning problem was pioneered by research in the memory allocation problem in the polyhedral model [29] [44] [25] [26] [13] [12] [6]. There is a long history of researchers using this model to build systems that
| App     | Pattern | System | Slice | BRAM | DSP |
|---------|---------|--------|-------|------|-----|
| denoise |         |        |       |      |     |
|         | Baseline| 303    | 4     | 0    |     |
|         | Spatial  | 330    | 4     | 0    |     |
|         | Ours     | 213    | 2     | 0    |     |
| deconv  |         |        |       |      |     |
|         | Baseline| 597    | 5     | 5    |     |
|         | Spatial  | 743    | 6     | 4    |     |
|         | Ours     | 532    | 3     | 0    |     |
| denoise-ur |     |        |       |      |     |
|         | Baseline| 795    | 8     | 0    |     |
|         | Spatial  | 1116   | 8     | 5    |     |
|         | Ours     | 659    | 6     | 0    |     |
| bicubic |         |        |       |      |     |
|         | Baseline| 238    | 4     | 0    |     |
|         | Spatial  | 309    | 4     | 0    |     |
|         | Ours     | 209    | 2     | 0    |     |
| sobel   |         |        |       |      |     |
|         | Baseline| 1523   | 9     | 9    |     |
|         | Spatial  | 1801   | 10    | 4    |     |
|         | Ours     | 1214   | 5     | 0    |     |
| motion-lv |       |        |       |      |     |
|         | Baseline| 425    | 4     | 0    |     |
|         | Spatial  | 1737   | 6     | 0    |     |
|         | Ours     | 187    | 3     | 0    |     |
| motion-lh |       |        |       |      |     |
|         | Baseline| 334    | 4     | 0    |     |
|         | Spatial  | 1333   | 6     | 2    |     |
|         | Ours     | 210    | 3     | 0    |     |
| motion-c |         |        |       |      |     |
|         | Baseline| 155    | 2     | 0    |     |
|         | Spatial  | 93     | 4     | 0    |     |
|         | Ours     | 69     | 2     | 0    |     |
| Avg.    | Baseline| -29.8% | -32.4%| -100%|     |
|         | Spatial  | -46.1% | -46.8%| -100%|     |

Table 2. Comparisons on Virtex 7

| App     | Pattern | System | LUT  | FF   | BRAM | DSP |
|---------|---------|--------|------|------|------|-----|
| denoise |         |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| deconv  |         |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| denoise-ur |      |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| bicubic |         |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| sobel   |         |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| motion-lv |       |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| motion-lh |       |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| motion-c |         |        |      |      |      |     |
|         | Baseline|        |      |      |      |     |
|         | Spatial  |        |      |      |      |     |
|         | Ours     |        |      |      |      |     |
| sw      | Fig 12a |        |      |      |      |     |
|         | Merlin   | 6983   | 7625 | 560  | 7   |
|         | Spatial  | 40286  | 48887| 396  | 21  |
|         | Ours     | 9485   | 3916 | 270  | 0   |
| spmv    | Fig 12b |        |      |      |      |     |
|         | Merlin   | 31224  | 11018| 256  | 0   |
|         | Spatial  | 93972  | 128015| 20  | 30  |
|         | Ours     | 12269  | 7359 | 6    | 0   |
| sgd     | Fig 12c |        |      |      |      |     |
|         | Merlin   | 9466   | 9255 | 504  | 38  |
|         | Spatial  | 21022  | 27725| 12   | 21  |
|         | Ours     | 4711   | 458  | 12   | 0   |
| Avg.    | Merlin   | -48.1% | -78.3%| -71.4%| -100%|
|         | Spatial  | -74.0% | -81.7%| -37.7%| -100%|

Table 3. Comparisons on AWS F1 (VU9P).

efficiently generate a memory allocation scheme that minimizes the overall memory footprint of an application. The process is focused on determining the live-ness of data in the program and determining a pseudoprojection mapping for memory references such that the program can be realized with a smaller memory footprint.

A related line of research involves computing memory partitioning schemes for distributed memory machines (DMMs) [27] [19] [34]. In these systems, the compiler builds a partitioning scheme of the program’s arrays such that the data is distributed across different parallel processors as efficiently as possible. The systems use a model that estimates the cost of communicating data between processors to determine the best way to generate the partitioning scheme.

Many SDH frameworks require the programmer to manually partition their arrays, but there has been recent work on developing tools that solve the problem automatically [42] [43] [10] [11] [22] using either hyperplane- or lattice-based techniques. Either technique can automatically solve the most common partitioning problems, but one may be more efficient for a certain problem than the other. Furthermore, one technique may have many solutions to the problem and estimating which one is the most cost-effective when synthesized and mapped to hardware can be tricky. These prior systems generally have first-order rules for determining which solution it finds is the most cost-effective one, but this does not always provide the best answer. Our system incorporates the advantages of both techniques by looking for both hyperplane and orthogonal lattice solutions. It also finds alternative parameters for calculating physical addresses,
specifically for hyperplane partitioning. It then uses built-in ML models for estimating the cost of each partitioning scheme when mapped to the underlying FPGA resources. It also applies mathematical transformations to the bank resolution arithmetic to generate circuits that are more efficient in hardware. This almost always removes the need for allocating DSPs to the bank resolution logic, which is generally the scarcest resource for algorithms that are accelerated by FPGAs.

6 Conclusion
As the compiler community continues to develop SDH tools that can express increasingly complicated loop scheduling and parallelization patterns, the problem of automatically partitioning memories quickly and efficiently is becoming increasingly important. We have introduced a banking analyzer that is scalable with programming models that are equipped to express complicated applications. Our system provides an appropriate solution to three challenges regarding the problem of memory partitioning: formulating constraints in a flexible SDH framework, quickly finding cheap solutions, and choosing the most cost-efficient one. We have trained a simple ML model based on decision trees and parallelization patterns, the problem of automatically partitioning memories quickly and efficiently is becoming increasingly important. We have introduced a banking analyzer that is scalable with programming models that are equipped to express complicated applications. Our system provides an appropriate solution to three challenges regarding the problem of memory partitioning: formulating constraints in a flexible SDH framework, quickly finding cheap solutions, and choosing the most cost-efficient one. First, our system is capable of producing a set of banking constraints regardless of the programmer’s choices in loop scheduling, access patterns, and parallelization such that it can guarantee the correctness of its partitioning solution. Next, we take advantage of the fact that the bank resolution logic is fully exposed in the IR to implement IR transformations aimed at nodes prevalent in banking resolution arithmetic. We use heuristics to steer the analyzer towards solutions that can be optimized in hardware. Finally, we have trained a simple ML model based on decision trees to quickly estimate the resource utilization of laying out various banking schemes in hardware. The model is accurate enough to select the best scheme.

Overall, all three of these features are necessary to have a practical solution for automatically partitioning memories in the compiler. Our system is capable of reducing LUT utilization by 86% and BRAM utilization by 38% over a naive system for a variety of complicated, real-world benchmarks. It almost always eliminates all DSP usage from the banking arithmetic for stencil and complex access patterns. For problems with massive solution spaces, it can cut the time spent searching for solutions in half.

References
[1] 2016. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[2] 2017. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.
[3] 2019. SDAccel Documentation. https://www.xilinx.com/support/documentation/navigation/dev-tools/software-development/sdacell.html.

arXiv Preprint, Work in Progress, Palo Alto, California

[4] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. 2018. Accelerating CNN inference on FPGAs: A Survey. ArXiv abs/1806.01683 (2018).
[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In International Conference on Compiler Construction. Springer, 283–303.
[6] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. Automatic Storage Optimization for Arrays. ACM Trans. Program. Lang. Syst. 38, 3, Article 11 (April 2016), 23 pages. https://doi.org/10.1145/2845078
[7] K. J. Brown, H. Lee, T. Romp, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 194–205.
[8] Salvatore Cardamone, Jonathan Kimmitt, Hugh Burton, Timothy Tomman, Shurui Li, Wayne Luk, and Alex Thom. 2019. Field-programmable gate arrays and quantum Monte Carlo: Power efficient coprocessing for scalable high-performance computing. International Journal of Quantum Chemistry 119 (02 2019), c25853. https://doi.org/10.1002/jqua.25853
[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. ACM, 785–794.
[10] Alessandro Cilardo and Luca Gallo. 2015. Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning. ACM Trans. Archit. Code Optim. 11, 4, Article 45 (Jan. 2015), 25 pages. https://doi.org/10.1145/2675359
[11] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. ACM Trans. Des. Autom. Electron. Syst. 16, 2, Article 15 (April 2011), 25 pages. https://doi.org/10.1145/1929943.1929947
[12] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. 2016. Extended Lattice-based Memory Allocation. In Proceedings of the 25th International Conference on Compiler Construction (CC 2016). ACM, New York, NY, USA, 218–228. https://doi.org/10.1145/2892208.2892213
[13] A. Darte, R. Schreiber, and G. Villard. 2005. Lattice-based memory allocation. IEEE Trans. Comput. 54, 10 (Oct 2005), 1242–1257. https://doi.org/10.1109/TC.2005.167
[14] C. De Sa, M. Feldman, C. Ré, and K. Olukotun. 2017. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). 561–574. https://doi.org/10.1145/3079856.3080248
[15] David Durst, Matthew Feldman, Dillon Huff, David Akley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983
[16] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20, 1 (1991), 23–53.
[17] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. International journal of parallel programming 21, 5 (1992), 313–347.
[18] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. International journal of parallel programming 21, 6 (1992), 389–420.
[19] C. H. Huang and P. Sadayappan. 1992. Communication-free hyperplane partitioning of nested loops. In Languages and Compilers for Parallel Computing. Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–200.
