Minimizing Search Time and Cost of Partitioned Memories in Hardware*

Subtitle†

Anonymous Author(s)

Abstract

Abstract abstract Abstract abstract Abstract abstract Abstract abstract Abstract abstract ....

Keywords memory partitioning, loop fusion, hardware

1 Introduction

[MATT: What is the current state of high level programming for fpgas]

Making FPGAs programmable by high-level application developers has emerged as a popular research topic. People who want to program FPGAs must invest lots of time learning a hardware description language (HDL), such as Verilog or VHDL, which sits at a similar level of abstraction as assembly. The compiler community has recognized the value in having higher level languages that allow developers to be much more productive without as much experience in hardware design. However, many traditional RTL designers are slow to adopt these higher level languages over performance concerns. The developer often knows a lot about specific features in their application that cannot be expressed in a high level language, and therefore must trust that the compiler is making smart decisions about how to transform their source code under the hood. However, it is clear that a compiler can iterate through design choices and figure out optimal ways to transform the code beyond what a human is capable of. There is an urgent need to make hardware domain specific languages (DSLs) generate fully optimized code in order for them to gain traction in the community and assist developers map increasingly complex applications to FPGAs.

One feature of hardware DSLs is that they make it very easy to capture spatial and pipeline parallelism at a high level and allow the developer to make algorithmic design choices without being burdened by the down-stream impacts that these changes have on the physical hardware. One such impact is banking, which is how memories that serve multiple accesses are physically laid out to avoid port conflicts.

Unlike in software, memories expressed in hardware cannot serialize requests to DRAM, cache, or dynamic registers in an efficient and general way without dedicating a significant amount of resources to this network, so a compiler must statically determine how to partition and buffer a memory. From the developer's point of view, memories are logically indexed. It is the job of the compiler to statically allocate physical memories that compose the single logical memory. This allocation involves three components: partitioning, duplicating, and buffering.

While the algorithms for computing banking schemes has been well studied in recent years, computing the "optimal" scheme in the shortest amount of time is still an open question. In this paper, we outline our contribution to quick and efficient banking and buffering analysis as they relate to hardware DSLs. We begin by formalizing the concepts involved in computing banking schemes. We then discuss how using this paradigm for banking analysis, we can steer the compiler to discover cost-efficient banking schemes more rapidly than previous approaches. Next, we discuss how a mixture-of-experts inference network can be used to choose the cheapest scheme out of a few options. Finally, we implement these ideas in a hardware DSL and discuss the results.

*Title note

†Subtitle note

PL'18, January 01–03, 2018, New York, NY, USA

2018.

[MATT: see if there are any existing numbers in literature for baseline banking that other people use. Baseline is cong, then compare our old stuff, then our new stuff.]
[MATT: massage spatial into doing what cong does, show cases we handle and they dont]

2 Background

[MATT: good performance is often about the structuring of the memory. Smart structuirng of memories is what allows you to have performance. Match memory to the needs of the computation]

[MATT: Talk about jason cong banking math]

In the ideal world of purely affine accesses and polyhedral models, memory partitioning for HLS applications has been well studied. Here we summarize how banking is formalized in the context of our discussion, borrowing algorithms and ideas from previous work with a few extensions.

2.1 Iterator Dephasing

[MATT: ripped this section out of ppopp, make it fit somewhere here] Iterator dephasing is the consequence of body unrolling in a high level language. Banking is significantly easier to solve before unrolling the IR, but the banking analyzer must know a lot of details about exactly how the IR will be unrolled. The analyzer must start by simplifying the addresses of an access into bare affine components and lump un-analyzable components into "random" variables that are aware of what iterators they depend on. [MATT: example of outer product?] Whenever possible, the compiler should connect "broadcast" ports to a memory when it can guarantee that different lanes of the unrolled body will access the exact same address during the exact same cycle. This reduces the number of ports and logic required to instantiate a memory. In order to guarantee this, the compiler must be able to 1) perform retiming analysis to know where in the inner controller's schedule the accesses are meant to occur and 2) perform lockstep analysis to decide whether the iterators in the affine access pattern will increment in lockstep with one another when they are derived from different lanes.

[MATT: maybe enumerate the rules for lockstepness here? Will look mathy, but may take a lot of space]

When a user writes code, they should have the tools necessary to understand whey the compiler viewed certain iterators as incrementing in lockstep or not. This will help them refactor their code in a way that preserves lockstepness and results in lower resource utilization. Even when iterators dephase relative to one another, banking analysis still has the opportunity to find a valid hierarchical banking scheme, such that it can separate certain lanes as "non-interfering" in a dimension that is out of view, even if they interfere relative to the dimension that is currently in view. [MATT: show example of this?]

2.2 Banking Scheme

The "banking scheme" of a memory is described by two sets of parameters, primitive and derived. **Primitive parameters** are those that can be fed into polytope emptiness test, along with information about the individual lanes of all accesses to the memory, to determine whether the scheme has bank conflicts or not. **Derived parameters** are those that are computed as a function of the primitive parameters and are required in order to resolve logical address requests to the memory into physical bank and offset values to dispatch to BRAM resources. A simple way to think about these sets of parameters are that primitive parameters control which banks a logical address will touch, and the derived parameters further resolve how the logical address can be converted into a physical address for the exact bank that it touches.

[MATT: do I need to describe how to construct the emptiness matrix sent to isl?]

Primitive parameters for an n-dimensional memory consist of:

- N = Number of banks. Vector containing either 1 or n values. While values between 1 and n are possible, they tend to add unnecessary complexity that is not justified by the efficiency of these schemes.
- B = Blocking factor. Vector with same size as N
- alpha = Partition vector. Vector of size n.

[MATT: I think hierarchical banking isn't really talked about in the original banking papers? maybe discuss in detail]

From these values, a banking address can be computed. The banking address is either 1-dimensional or n-dimensional. Given logical address $\vec{x}$, bank address is

$BA = \lfloor \frac{\vec{x} \cdot \vec{alpha}}{B} \rfloor mod N$

or

$BA_i = \lfloor \frac{x_i * alpha_i}{B_i} \rfloor mod N_i$

A useful way to think about these values is that the partition vector describes how to increment the bank upon taking one step in a particular dimension. N describes how many banks there are before wrapping back to 0. B is a scaling factor that allows strange access patterns (i.e. x(i) + x(i+1) where i steps by 3 OR x(i) + x(16) where i goes from 0 to 15) to pass polytope testing with fewer banks by squeezing addresses into the same banks.

[MATT: mention co-primeness properties of these parameters?]

[MATT: maybe I should point out n-dimensional N can be rewritten as 1-dimensional N (I think this is true, but not sure). But a hierarchical view sometimes helps a lot with speeding up the banking search]

Derived parameters for an n-dimensional memory consist of:

- P = Bank offset grouping. n-dimensional vector which scopes a "neighborhood" of the logical memory in which each bank exists exactly once* (* if B = 1).
- Pad = Amount to pad each dimension. n-dimensional vector required if abutment of "neighborhoods" does not evenly divide the memory
- duplicity = Number of elements each bank contains in a "neighborhood," for B > 1 cases
- darkVolume = Amount of volume that is mathematically inaccessible due to any duplicity > 1.

P is computed by finding the periodicity of the bank address equation per dimension of the memory. The periodicity can be computed as $N * B/gcd(N, alpha)$. [MATT: keep running into this thing where the equation is different for n-d banking and flat banking. It may take up lots of space...] Then, each periodicity is expanded into a set of all of its divisors. Given this list of lists, every possible list which chooses one from each set and whose product equals the number of banks is tested to see if it contains each bank exactly once (or at least once for B > 1). The list which satisfies this requirement and minimizes darkVolume and padded volume is chosen.

Using these derived parameters, a scalar value for bank offset can be computed as:

$BO = \sum_0^n \lfloor \frac{x_i}{P_i} \rfloor * \sum_0^i \frac{dim_i + pad_i}{P_i} * maxduplicity + \sum_0^n x_i * B_i$

[MATT: Equation is probably wrong. Transcribed in a hurry and from memory]

A useful way to think about this bank offset equation (for B = 1) is that the logical memory is divided into non-overlapping neighborhoods, where every address in the neighborhood has a different bank address but same bank offset. The equation determines which neighborhood the address falls in and converts this to an offset. Padding is required so that at synthesis time, physical BRAMs are allocated enough volume to contain the very last neighborhood. If B > 1 for any dimension, the same strategy is applied except a modulo correction is introduced to uniquely assign elements of the same bank to different offsets. Not all banks have the same level of degeneracy within a neighborhood, which causes darkVolume as this inaccessible volume must be allocated during synthesis for the equation to work properly. [MATT: Still need to prove why the extra mod correction works without collisions?]

While there is a more complicated bank offset equation that involves a lot of muxing for placing a logical address to a unique physical address without padding and darkVolume, we decided that the extra hardware complexity is not worth the reclamation in resources. [MATT: is this enough about why we chose the bloated volume, simpler math rather than leaner volume, complicated math?]

2.3 Access Information

In addition to the banking scheme parameters outlined previously, each lane of each access has its own parameters that describe how it interacts with the given scheme. Specifically, this is the "switching factor" of the access. For purely affine accesses where the iterators' start, stop, and steps are statically known, the banks that this pattern can "see" can be enumerated by computing the residual set of the bank address equations. Specifically, the residual set can be summarized by a "residual generator" with parameters A, B, and M described below:

$A = gcd(alpha_i, N_i)$

$B = min(x_i * alpha_i)$

$M = N_i$

$SF = number of unique values in A*k+B mod M for all integer k$

[MATT: also did this one hastily and from memory. Should the equation for B > 1 RG be]

It is important to note that a large switching factor for readers is more expensive than a large switching factor for writers. This is because a reader requires a crossbar to connect the address port to the correct physical memory, as well as a crossbar to serve the output data back to that reader. A writer, on the other hand, can be implemented simply as a write-enable mask that is broadcasted to all banks within its residual generator set.

This information will be incorporated into cost analysis, as described later in this paper

2.4 Examples

Here we discuss two concrete examples to demonstrate the interplay between primitive and derived parameters.

[MATT: Is spatial, pseudocode, or affine access patterns the best way to show this? Also need to include picture]

Example 1:

x = 6x8 memory

read x(i,j), x(i+1,j), x(i,j+1), x(i+1,j+1), where i,j step by 2

One of many valid primitive parameter options here is:

- N = 4
- B = 1,1
- alpha = 1,2

These values lead to bank periodicities of 4,2, which expands to ((1,2,4),(1,2)) and generates potentially valid Ps of (2,2) and (4,1). The P=(2,2) results in padding of 0,0 so it will be chosen.

- P = 2,2
- pad = 0,0
- degeneracy = 0 -> 1, 1 -> 1, 2 -> 1, 3 -> 1
- darkVolume = 0

[MATT: pictures of bank(color) + offset(number) view of this scheme]

The switching factor of each lane is 2, as $1 * 2 * k_0 + 2 * 2 * k_1 mod 4$ can resolve to 2 values.

Example 2:

x = 6x9 memory

read x(i,j), x(i+1,j), x(i,j+1), x(i+1,j+1), where i steps by 2, j steps by 3

One of many valid primitive parameter options here is:

- N = 2,2
- B = 1,2
- alpha = 1,3

These values lead to bank periodicities of 2,3, which expands to ((1,2),(1,3)) and generates only one valid P of (2,3) [MATT: aww thats boring. Should I put a more fun example?]. The P=(2,3) results in padding of 0,0. The duplicity, however, is such that column bank 1 exists twice in each neighborhood, and column bank 0 exists only once. This results in a darkVolume of 2 for each neighborhood (BA=0,0 has 1 and BA=1,0 has 1), and there are 9 neighborhoods.

- P = 2,3
- pad = 0,0
- degeneracy = 0,0 -> 1, 1,0 -> 1, 0,1 -> 2, 1,1 -> 2
- darkVolume = 18

[MATT: pictures of bank(color) + offset(number) view of this scheme]

The switching factor of each lane is 1, as $1 * 2 * k_0 mod 2$ and $3 * k_1/2 mod 3$ both resolve to only one value.

2.5 Host Language

[MATT: Should these paragraphs be removed because they are boring and dont add to the story?]

Spatial is a high level DSL embedded in Scala that is built on arbitrary nesting of parallel patterns and explicit memory declarations that are amenable to hardware. It has an IR that makes it easy to do polydedral analysis on memory accesses. Because it is embedded in Scala, it is straightforward for the language to generate Chisel, which is an RTL language that is also embedded in Scala. Chisel allows for parameterized hardware templates and straightforward stitching of hardware blocks with each other. In order to assess our results, we chose to rewrite the banking analysis in Spatial to incorporate the tweaks and enhancements discussed in this paper.

Now that we have introduced what parameters are required to define a banking scheme, we spend the rest of the paper discussing 1) how to prune and traverse the search space in such a way that improves the likelihood of finding valid banking schemes quickly, and 2) how to compare the schemes against one another to choose the most resource-effective one. We also provide a few ideas for how to embellish a high level DSL in such a way that an educated user can have a significant amount of power in steering the banking system in a favorable way.

3 Search

s Here we formalize the five characteristics that describe a set of potential banking schemes the compiler must test to find a

```
1   // Hierarchical required
2   for (i <- 0 until 10 par 2)
3     for (j <- f(i) until 10 par 2)
4       ... = x(i, j) // j from i = 0 is not
            lockstep with j from i = 1
5
6   // Hierarchical needs N=6,4 (=24) banks, flat
        needs 6 banks with alpha = 1,1
7   for (i <- 0 until 12, j <- 0 until 16 par 4)
8     y(i,j) = ...
9   for (i <- 0 until 12 par 6, j <- 0 until 16)
10      ... = y(i, j)
```

Figure 1. Flat vs Hierarchical examples*m*.

valid scheme. The goal in this section is to describe how partitioning the entire banking search space into these categories can significantly improve the search time. [MATT: find if there is prior work mentioning how to speed up search time]. As discussed later, these properties lie at the core of what the user should be allowed to annotate in order to steer the banking analyzer towards better solutions more quickly.

- Flat or Hierarchical
- N-strictness
- Alpha-strictness
- Axis Regrouping
- Block Cyclic

[MATT: provide one or two interesting banking cases to drive this section and explain why multiple setups can provide seemingly good schemes]

Flat and hierarchical refer to the dimensionality of the bank addressing. Flat refers to the case where N and B are scalars. Hierarchical refers to the case when N and B are vectors of size n. The reason both should be treated differently is that there are some schemes which are much easier to analyze with hierarchical schemes, and others are much easier to analyze with flat schemes.

N-strictness and alpha-strictness refer to the numeric properties of the potential N's and alpha's a certain category has. The best banking scheme is generally one in which all unrolled lanes have a switching factor of 1. After accesses are grouped and broadcasting reads are filtered out, the only way to directly connect banks is if there is one bank for each lane in the group. Therefore, N's that are expected to result in the lowest resource utilization and satisfy emptiness constraints the fastest are those that are related to the number of lanes and dimensions of the memory. The first set of Ns that should be tried are those that are either divisors of the number of accesses in a particular group, or those that are divisors of a dimension of the memory. Values related to the dimension of the memory may be non-intuitive at first, but it is possible for such a scheme to result in a switching factor of 1 without introducing padding for that dimension.

If switching factor of 1 is not possible, Ns that are powers-of-2 are likely to be the next best schemes, as multiplies and

```
1        // Duplication per row required to guarantee
             safety
2        for (i <- 0 until 10 par 2, j <- 0 until 10)
3        ... = x(f(i), j) // f(i_0) and f(i_1) may be
             equal
4
5        // Duplication by col makes 15 copies with N=4
             each, which COULD be cheaper than 1 copy
             with N=60 (non pow2 xbars + resolution math
             ), depending on size of mem
6        for (i <- 0 until 16 par 2, j <- 0 until 30 par
              15)
7          y(i,j) = ...
```

Figure 2. duplication examples *m*.

divides can be rewritten as bit shifts, and modulo can be rewritten as bitwise 'and'. The one exception is when a non-power-of-2 N results in a significantly smaller mux, which is why it is important to provide the user with the ability to steer banking. Finally, if no NBest or NPow2 schemes are found, any values not covered by these two categories should be exhaustively tested.

Alpha-strictness follows the exact same properties as N-strictness, with the only difference being that potential alphas are bounded by a particular attempted value of N. I.e. it does not make sense to test alphas which are greater than the particular instance of N. We emphasize that choosing values from the set of divisors of each dimension is often a good strategy to find a valid banking scheme quickly.

Axis regrouping refers to which logical dimensions of the memory the compiler should bank-by-duplication. For example, if there is a 2D memory where two random rows are required in parallel, but the column addresses are analyzable, the compiler should be allowed to make two copies of the memory, one for each row, and bank column-wise for accesses in that dimension. In other cases, depending on the dimensions of the memory and the parallelizations relative to each dimension, it could be cheaper in terms of resources to duplicate rather than bank. Note that even Flat banking schemes can have axis regrouping on a subset of dimensions because the analysis must always bank for all writer lanes in order to guarantee data consistency. It can, however, bank-by-duplication for different read lanes, and having the power to express this inside the compiler is a powerful tool for finding cheap banking schemes faster.

Block cyclic enumerates the possible values of B that the compiler should investigate. In most cases, B = 1 results in the best solution, however there are some cases when B > 1 should be chosen. The compiler should treat these cases differently, and first exhaustively search solutions for B = 1 before attempting solutions for B > 1 unless the user specifies otherwise. Certain access patterns that require B > 1 result in physical addresses that are inaccessible by the logical addressing in the source code, and therefore may be memory inefficient. However, B > 1 can sometimes result in significantly fewer banks overall, which could be a cheaper solution overall [MATT: i.e. Kmeans]

[MATT: possibly show a chart displaying search time in ms for a common access pattern to find a valid banking scheme under each of these directives. Maybe introduce why directives are useful here. Also maybe talk about the banking/decisions.html here]

4 Cost

In the previous section, we describe how and why the compiler computes various valid banking schemes. We previously focused on how to make small tweaks in the compiler to significantly reduce the time spent searching for valid banking schemes. The next problem is to quickly estimate the resource cost of each one and choose the best one.

Given the variety of directives, such as Flat vs Hierarchical, Axis Duplication, and Block Cyclic, it is not always straightforward to compare the costs heuristically. One must consider the cost of the memory partitioning itself (i.e crossbar costs, BRAM fragmentation costs, etc.) as well as the bank resolution arithmetic cost (i.e. pow2 operations vs arbitrary constants)

For example, sliding window convolutions are an especially popular kernel that many people are interested in implementing on FPGAs, but there are often no valid schemes with direct bank connections. There are [MATT: example?] domains which have memories that may have some schemes containing many banks and small crossbars on each lane, or fewer banks but very wide crossbars for each lane.

The first step in comparing the valid banking schemes against each other is to look ahead and determine what nodes will be created by the compiler's unroller and access conversion step. Specifically, the banking analyzer must be able to statically observe the residual generators of the modulo arithmetic used to compute bank address, as well as the arithmetic nodes created by bank offset calculation. Spatial handles this by maintaining metadata about the iterators in the IR and using this metadata assist in rewrite rules that happen after unrolling. [MATT: Wrote this on the fly, should be heavily revised to sound smart and accurate].

The next step is to feed these nodes, in addition to the SRAM instantiation nodes, into a model to predict the LUTs, Regs, BRAMs, and DSPs consumed. We choose to use a machine learning pipeline to obtain the resource numbers instead of running an FPGA toolchain or estimating the resource utilization based on some hand-written heuristics. First, an FPGA toolchain usually takes a few hours to run. However, we expect a compiler with the banking optimizer to complete within a few minutes. Second, an FPGA toolchain is a blackbox containing a large set of preset rules. Without assists from the FPGA vendors, it is not possible to accurately model the behavior of the toolchain with simple heuristics.

Memory instantiation nodes are explicit in the Spatial IR and fully parameterized. Hence, we can extract training data based on both real-world applications and contrived applications that steer the compiler to choose some of the more obscure banking schemes.

Previous work (dk, jc) uses deep neural networks to predict the resource utilization. However, deep models run slowly on a CPU and usually cannot meet the time budget when embedded in a compiler pass. In addition, deep models tend to overfit on the training set created by a compiler. In this work, we use a tree-based ensemble model called the graident boosting tree. We first use the gradient boosting method to iteratively generate weak models. Specifically, at each iteration, a model is created to fit on the training data. At the next iteration, a new model is created to optimize the residual of the previous model. After all the iterations complete, we ensemble the weak models with a random forest tree. Each weak model is very light-weight. As a result, the ensembled model consumes a very small portion of a compiler's runtime. Moreover, the ensembling approach helps reduce the issue of overfitting.

[TIAN: Prediction results over here: MAE, MSE, R2, Prediction Interval]

5 APIs

[MATT: possibly use this section to talk about the api that should be exposed for speeding up banking and making it more efficient, as alluded to in previous sections.]