Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software

Yu Ting Chen and Jason H. Anderson
Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario
Email: joyuting.chen@mail.utoronto.ca, janders@ece.toronto.edu

***Abstract*—Some modern high-level synthesis (HLS) tools [1] permit the synthesis of multi-threaded software into parallel hardware, where concurrent software threads are realized as concurrently operating hardware units. A common performance bottleneck in any parallel implementation (whether it be hardware or software) is memory bandwidth – parallel threads demand concurrent access to memory resulting in contention which hurts performance. FPGAs contain an abundance of independently accessible memories offering high internal memory bandwidth. We describe an approach for leveraging such bandwidth in the context of synthesizing parallel software into hardware. Our approach applies trace-based profiling to determine how a program's arrays should be *automatically* partitioned into sub-arrays, which are then implemented in separate on-chip RAM blocks within the target FPGA. The partitioning is accomplished in a way that requires a single HLS execution and logic simulation for trace extraction. The end result is that each thread, when implemented in hardware, has exclusive access to its own memories to the extent possible, significantly reducing contention and arbitration and thus raising performance.**

Keywords*—*High-Level Synthesis, memory architecture, multi-threaded HLS.

I. Introduction

High-level synthesis (HLS) is becoming a prevalent design methodology for FPGAs [5], [11] with the introduction of tools such as Vivado HLS from Xilinx [15] and Altera's OpenCL SDK [8]. HLS is an active area of research with significant effort being put towards closing the performance, power and area gap between HLS-generated hardware and human-expert-designed hardware. In any hardware computing system, memory is a frequently occurring culprit for limited performance; that is, memory bandwidth and contention for memory among multiple accessors are common roots of limited computational throughput. Processors, and the software programs that run on them, are typically based on the assumption of a single unified memory address space. FPGAs, on the other hand, contain many small memories, distributed spatially, that can each be accessed concurrently. The 3 approaches for increasing memory bandwidth are *i)* memory *replication*, *ii)* implementing *multi-ported* memories and *iii)* memory banking through *partitioning*. Memory replication may incur a high area cost and would require a synchronization mechanism across multiple copies of the same memory unless the data is read-only. Increasing the number of memory ports is expensive and may not be possible on the FPGA since the block RAMs have limited port configurations. In this article, we consider automated generation of banked on-chip memory architectures in the high-level synthesis of parallel software programs into parallel FPGA hardware, with the goal of reducing memory contention and raising bandwidth.

One of the primary aims in pursuing a hardware implementation vs. a software implementation is to exploit spatial parallelism in the hardware. Towards this, some modern high-level synthesis (HLS) tools, such as the LegUp HLS framework from the University of Toronto, provide spatial exploitation capability at the software-specification stage by supporting the synthesis of parallel software (threads) into parallel hardware [2]. The parallel software threads in multi-threaded `C` programs written using the Pthreads or OpenMP standards, are realized as concurrently operating units in hardware. In this work, we are concerned with the synthesis of memory architectures for such spatially parallel implementations, where concurrently operating hardware units may vie for access to the same memory ports at the same time. We seek to alleviate such contention through automated synthesis of banked architectures such that hardware units have (partially) exclusive access to particular ports to the extent possible.

We implement our work in the LegUp HLS framework within its Pthreads/OpenMP parallel flow. We first augmented LegUp with user-constraint-driven array partitioning, similar to that available in Xilinx's Vivado HLS tool. Based on user-supplied directives, the new functionality permits arrays to be partitioned in a variety of different ways by the compiler, where each partition is then implemented in separate RAMs within the FPGA implementation. However, two issues exist with the constraint-driven approach: 1) it can be difficult to know which style of array partitioning is best for a given array as it depends on the memory access patterns of the parallel program and also the architecture of the target FPGA, and 2) specifying array partitioning directives is non-standard, and we believe it to be particularly difficult for software engineers with no knowledge of FPGA hardware.

In light of these two issues, we propose a memory-trace profiling-based approach that automatically determines array partitionings to realize banked memory architectures offering low memory contention and high bandwidth for parallel spatial hardware as produced by LegUp's Pthreads/OpenMP flow. An advantage of the proposed approach is that it requires a single HLS execution and hardware simulation of the generated hardware to extract a memory access trace, as opposed to more exhaustive approaches requiring many synthesis/simulation runs. Specifically, we have developed a memory-trace simulator that permits rapid evaluation of various memory partitioning schemes (banking architectures) using a single execution trace of the hardware with unpartitioned arrays

through modeling contention and arbitration.

The key contributions of this paper are:

- Infrastructure for array partitioning implemented in the open-source LegUp-HLS framework.
- An automated trace-based array partition scheme detection approach using a lightweight memory simulator with the overall goal of reducing execution cycles in HLS-generated hardware for multi-threaded software.
- Automated arbiter insertion and support for cases where "exact" thread-driven partitioning is not possible, for example, for scenarios wherein two threads *occasionally* access the same RAM.
- An experimental study demonstrating the efficacy of the automated partitioning technique in comparison with a brute-force approach.

II. Background

A. LegUp HLS Tool

The LegUp HLS tool is implemented within the open-source LLVM compiler [9]. Within LLVM, the program is represented in the compiler's intermediate representation (IR), which resembles RISC assembly code, being composed of simple instructions such as multiply, add, branch, jump, etc. Prior to LegUp HLS, the program's IR is subjected to LLVM's compiler optimization passes, such as dead-code elimination, loop rotation, and common sub-expression elimination. LegUp takes the optimized IR as input and performs the traditional HLS stages [6] of resource allocation, scheduling, binding and Verilog generation. In this work, array partitioning for the purposes of banked memory architecture synthesis is implemented as a generic LLVM optimization pass. As will be detailed below, the original arrays in the IR are broken into pieces and array accesses are steered to the appropriate sub-array.

B. Synthesis of Parallel Software to Hardware

In LegUp's Pthreads synthesis flow, multi-threaded software is synthesized into parallel hardware whose behaviour closely matches the software semantics. Each software thread is synthesized into an instance of a hardware unit. Pthreads library function calls in the original `C` program are replaced with LegUp-specific wrapper functions that cause specific hardware structures to be produced in the RTL generation phase. For example, calls to `pthread_create` are translated into FSM functionality that invokes a hardware unit corresponding to a software thread. Support is provided for common thread synchronization approaches including mutexs (for critical sections) and barriers. Ultimately, the Pthreads flow permits a wide range of spatial parallel implementations to be realized through software changes alone. The interested reader is referred to [2], [3] for complete details. Our work here seeks to alleviate a specific (and common) performance bottleneck in such parallel hardware, namely, when multiple parallel hardware units contend for memory.

C. LegUp's Memory Architecture

LegUp HLS implements each array structure in a separate logical RAM. Depending on its size, the logical RAM may be implemented in multiple physical block RAMs in the FPGA implementation (e.g. when the array size exceeds the SRAM block size in the FPGA fabric). For RAMs that may be accessed concurrently by parallel hardware, LegUp automatically inserts arbitration circuitry. The arbiter permits single-cycle access in the absence of contention. Under contention, it is resolved in a round-robin style, in which case a hardware unit will stall until it is granted access, degrading performance. Such degradation is particularly common in the Pthreads synthesis flow.

Consider, for example, parallelized vector addition, $Z = A + B$, where Z, A, B are n-element arrays. In a typical multi-threaded implementation, each thread would operate on a portion of the original arrays. With t threads implemented as t parallel hardware units there would be significant contention on the ports of RAMs holding the arrays, even if the RAMs were in dual-port mode.

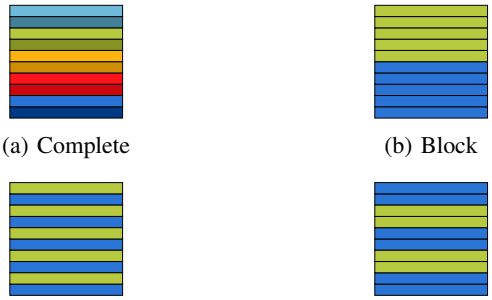
LegUp uses "points-to" analysis in LLVM to designate arrays as either *local*, *shared-local* or *global* [3]. Local arrays are accessed by a single function in the `C` code. Shared-local arrays are accessed by a limited number of functions, as determined statically at compile time. For arrays designated as global, the points-to analysis was unable to statically determine the accessors. In hardware, such arrays are implemented behind a global memory controller that uses a unique tag for each array to steer accesses to the correct SRAM block containing the array [1]. For this work, we focus on shared-local arrays since global memory requests cannot be parallelized due to the memory controller and local memory will be scheduled such as to avoid port contentions.

III. Related Work

A number of recent works have considered memory banking in HLS. Below we highlight those we believe are most relevant to the present research. However, an important distinction of our work is the focus on the memory architecture synthesis in the multi-thread HLS context. Our technique specifically targets the synthesis of banked architectures that result in reduced contention among the hardware implementation of parallel software. To the authors' knowledge, no prior work has considered this scenario.

As with the present work, a recent work by Zhou et al. [16] applied a trace-based approach to memory banking and employed a conflict-graph-based approach to map memory addresses to banks, with emphasis on multiplexer size minimization. Formal techniques were used to verify the mapping was indeed *conflict free*. As opposed to our approach, which chooses an architecture from the trace, [16] requires the number of banks as an input to the algorithm, and solely handles the conflict-free cases (there is no arbitration).

Other works include [4] which applies mathematical techniques to the memory partitioning problem. A polyhedral model is used to represent memory access patterns in loops and legal code transformations, and an integer lattice approach is used for memory partitioning, where the objective is to: 1) minimize contention, and 2) reduce "waste" (unused portions of the partitions). [10] targets accesses of multi-dimensional arrays in loop bodies and proposed a closed-form linear mapping from array accesses to bank indices to achieve zero conflicts (and II=1 for loop pipelining); subsequent optimizations then



(a) Complete (b) Block

(c) Cyclic (d) Block Cyclic

Fig. 1: Partitioning schemes applied to 10×10 matrix in the row dimension.

reduce the bank count and space wastage. [13] and [12] show static analyses which look for hyperplanes in memory accesses in loop bodies. The hyperplanes are described by a vector and a linear transformation translates between accesses to the original multi-dimensional array into accesses in the newly partitioned banks. However, [7] shows that the hyperplane solution does not always work due to a phenomenon known as bank switching, wherein the output of one bank may be required by multiple hardware accessors. Again, these works are not centered on multi-threaded software, but rather, partitioning of arrays accessed multiple times in a loop body.

The application of geometric (e.g. polyhedral) representations of memory accesses in a loop body to the memory architecture synthesis problem are not directly transferable to the related problem in the multi-thread memory contention context. The static approaches presented focus on finding conflict-free mappings, however, our approach allows for non-perfect partitions.

IV. Array Partitioning Implementation

A. Available Partitioning Schemes

There are 4 supported partitioning schemes for multi-dimensional arrays: *complete, block, cyclic,* and *block cyclic* shown in Fig. 1. These are similar to the schemes that are supported by Vivado HLS. In the complete scheme, each array element is designated to a separate partition. In the block scheme, partitions represent contiguous pieces of the original array. The cyclic scheme is an interleaved approach, where array elements a fixed distance apart are allocated to the same partition. Block cyclic combines the block and cyclic schemes.

Partitioning is available in any single dimension of a multi-dimensional array. For the purposes of this work, our focus is on globally declared data structures – i.e. those which are not locally scoped and therefore, can be shared between multiple functions or threads of the same function.

The specific partitioning scheme is based on several terms:

- **Number of Partitions** (n): Number of total partitions to be created.
- **Block Size** (b): Size of contiguous elements in the dimension of partitioning that constitute the partition.
- **Size** (S): Number of elements of the multi-dimensional array in the dimension of the partitioning.

B. Partitioning Implementation

Memory partitioning was implemented as a compiler pass in LLVM, where the input is the original IR and the produced IR is changed such that memory accesses to the affected arrays are modified to access from the sub-arrays. The pass takes advantage of the predicated load/store operations supported by LegUp. Predicated loads/stores differ from regular load/store instructions in that the memory enable signal is no longer only FSM-state dependent, but also dependent on a precomputed predicate signal. Predicates are typically the true/false outcome of a comparison. Leveraging the predicated load/store, the array partitioning pass can, for each memory access to a partitioned array, create multiple predicated memory instructions for each newly created partition, where the predicate is a compare instruction of the original index. The compare instruction (`icmp eq`) will evaluate to `true` only for one of these instructions at run-time. Therefore, the array partitioning transformation does not require extra memory bandwidth.

It is worthwhile to explain the detailed behavior surrounding predicated memory accesses, arbitration, and partitioning. Consider a four-thread program, where initially all threads access a single array. When synthesized to hardware, contention and delays due to arbitration may arise because the array, implemented as a logical RAM in the FPGA, has four accessors and at most two ports.

To resolve the contention, assume the array is partitioned into four pieces, where each thread accesses the partitions in a near-exclusive fashion, for example, each thread accesses one partition 95% of the time, but 5% of the time accesses one of the other three partitions. In such a scenario, the hardware implementation of each thread does indeed issue a predicated access to all four partitions; however, it is *only* the access for which the predicate is true that is sent to the arbiter. That is, the accesses associated with false predicates do not create contention in our implementation.

C. Limit to Power-of-2 Partitions

In order to access a memory location in the newly partitioned memory scheme, the new access will require both the new partition number, as well as a new index into the partition where memory location now resides. The equation to determine the new partition number, p_d, is:

$$p_d = \lfloor i_d \ / \ b \rfloor \ \% \ n \tag{1}$$

where i_d is the index of the access in the dimension of partitioning, d; b is the block size and n is the number of partitions.

The equation to compute the new index, new_i_d, into the partition is:

$$new\_i_d = (i_d \ \% \ b) + \lfloor i_d \ / \ (b \times n) \rfloor \times b \tag{2}$$

The indices into the other dimensions of the array remain unchanged since the other dimensions were not partitioned.

The above equations hold for the general case of *block cyclic* partitioning. They can be simplified in the case of *complete, block,* and *cyclic*. For example, notice that in the case of *complete* partitioning, b is 1 and n is S, the size of the

Scheme	Number Partitions	Block Size
Complete	$N_{cmp} = \{n \mid n = S\}$	$B_{cmp} = \{b \mid b = 1\}$
Block	$N_b = \{n \mid 1 < n < S\}$	$B_b = \{b \mid 1 < b < S, b \in \{2^i \mid i \in \mathbb{N}\}\}$
Cyclic	$N_c = \{n \mid 1 < n < S, n \in \{2^i \mid i \in \mathbb{N}\}\}$	$B_c = \{b \mid b = 1\}$
Block Cyclic	$N_{bc} = \{n \mid 1 < n < S, n \in \{2^i \mid i \in \mathbb{N}\}\}$	$B_{bc} = \{b \mid 1 < b < S, b \in \{2^i \mid i \in \mathbb{N}\}\}$

TABLE I: Partition configuration set of allowable value ranges for number of partition, and block size parameters.

array in dimension d of *complete* partitioning. Therefore, the equation to determine the new partition number is simply,

$$p_d = i_d \tag{3}$$

and the new index is simply equal to 0.

Observe that the above equations heavily rely on the division and modulo operators. These operations are potentially costly from both the performance and area perspectives in LegUp-generated hardware. For example, in the division/-modulo module LegUp instantiates, the number of cycles required is proportional to the bitwidths of the division/modulo operands. This implies that the cycle-count penalty incurred to compute partition number and index can override any cycle-count reduction afforded by partitioning in the first place.

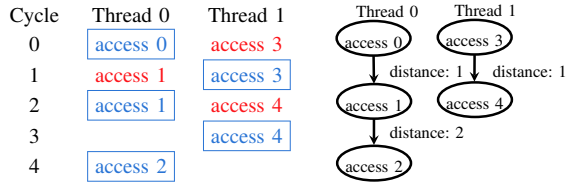
Consequently, we limit the solution space such that the division operations can be substituted by right shift operations and the modulo operations can be substituted by bit-wise logical-AND operations, which are both trivial in hardware. Table I shows the values of n and b that each partitioning scheme allows. There is only one possible setting for complete partitioning. In block partitioning, we can have any integer value greater than 1 and less than S number of partitions and our block size must be a power of 2 value greater than 1 and less than S, where S is the size of the array in the dimension of partitioning. In cyclic partitioning, we need a power of 2 number of partitions which is greater than 1 and less than S and a block size of 1. In block cyclic partitioning, we must have a number of partitions and block size which are both power-of-2 integers greater than 1 and less than S. Section V-D presents the solution space for a general multi-dimensional array.

V. Automatic Array Partitioning

A. Overview

At a high level, the automated array partitioning is implemented as follows: First, the multi-threaded software program, without any partitioning applied, is synthesized to hardware using LegUp HLS. The generated RTL is then simulated with ModelSim under typical input vectors, and a memory access trace is extracted, where, for each array/memory access, the address and the ID of the thread making the access is tracked.

Following the trace generation, the trace itself, as well as a *hypothetical* array partitioning scheme are inputted to our memory simulator, which, for the proposed scheme, estimates its performance benefits. With the single HLS execution and trace generation, we are able to assess the consequences of various partitioning schemes. And, by simulating a variety of different partitioning schemes, an optimal memory banking architecture can be determined. We elaborate on the steps in the subsections below.



(a) Thread access schedule (b) Memory access chain

Fig. 2: Memory access schedule of 2 threaded execution to memory access chain.

B. Memory Trace Collection

We modified the RTL-generation of the LegUp HLS tool to produce the relevant trace data in ModelSim simulation. Specifically, a `$display` statement is inserted whenever a request is made to a shared memory since it can be easily detected through the `request_in` vector of the associated arbiter. The information attached to each shared memory request is the cycle of the initial request, the specific block RAM being requested, the port name, the function trying to access the RAM, the instance number in the case of multi-threaded functions, and the address of the request to be used later for determining possible partition schemes.

C. Memory Partitioning Simulator

The memory simulator is a program which aims to model the arbitration of requests to a particular port of a specific block RAM. The program is written using Python. The simulator will, based on any partitioning scheme and an input memory trace collected during the unpartitioned execution, determine the number of port contentions that will occur, as well as the cycle number in which the last memory request will be serviced.

The program first creates a representation of the unpartitioned execution memory accesses as a chain of nodes in which each node represents a memory access and the edge weights between the nodes represent the *distance*, i.e. the intrinsic number of cycles between accesses. The *distance* may not be the same as the number of cycles between accesses in an execution trace since the trace captures cycles lost to contention. Fig. 2 shows a series of scheduled accesses for a program running 2 threads, as well as the memory access chains that are created from the memory trace. The boxed accesses represent the requests granted in that cycle, while the unboxed accesses represent the accesses which stalled due to contention. As seen here, the cycle difference between when *access 0* is granted and when *access 1* is granted is 2. However, the distance will measure 1 cycle, because the distance captures the number of cycles between accesses in the absence of contention.

Once the memory access chain is gathered, the simulator will "execute" the trace, where the memory accesses are scheduled according to the partitioning scheme under test, since we know the order of all memory accesses by each requester, as well as the specific addresses being accessed, which can tell us which partition the access would belong to. The simulator has an in-memory model of the round-robin arbiter instantiated by LegUp and therefore, it is able to assess cycles lost to contention, i.e. a requester waiting for a "grant" from the arbiter.

Listing 1 shows pseudocode for simulating the scheduling of a memory access trace for a particular array partitioning configuration, `partitionConfig`. Our approach bears similarity to event-driven logic simulation. Here, `setQ` is a set of queues: one for each arbiter in the partitioning scheme being simulated. Each queue stores the pending memory requests for a port on a memory bank. Initially, all arbiter queues are initialized to empty.

Listing 1: Algorithm for simulating a memory partitioning scheme

```
1  setQ: current pending accesses for each arbiter
2  T: memory trace
3  for each Q in setQ {
4      Q = {}
5  }
6  cycle = first access cycle in T
7  while (!done) {
8      done = True
9      for each Q in setQ {
10         populateQ(Q, partitionConfig, cycle, T)
11         executeCycle(Q)
12     }
13     cycle++
14     for each Q in setQ {
15         if Q !empty || accesses in T {
16             done = False
17         }
18     }
19 }
```

We commence simulation at the cycle of the first memory access (line 6) and enter the `while` loop (line 7). For each arbiter present in the design (line 9), as specified by `partitionConfig`, we use the present cycle, as well as the trace `T` to populate the queue with active requests (line 10). The `populateQ` function (not shown) takes accesses from the trace `T` which occur at time `cycle`, and for each such access, uses knowledge about the partitioning scheme to create a request in a specific queue. Then, the `executeCycle` function (line 11) uses an internal arbiter model to determine the request in each queue which will receive the grant; this request is removed from the queue. This process continues until there are no more requests outstanding in any arbiter-request queues and when the trace is empty (line 15). Upon termination, the `cycle` variable holds the cycle index of the last-serviced memory request. Partitioning schemes which reduce this quantity are generally desirable.

D. Finding the Optimal Solution

We take a brute-force approach to finding the optimal partitioning solution. This means we iterate through all the possible partitioning schemes[1], while predicting the number of cycles needed in order to complete all the memory accesses collected in the trace, as well as the number of cycles associated with stalls due to memory contention.

To provide insight on the size of the solution space, the total number of partitioning schemes that may be applied to one multi-dimensional array is:

$$NUM_{total} = NUM_b + NUM_c + NUM_{cmp} + NUM_{bc} \quad (4)$$

where NUM_b, NUM_c, NUM_{cmp}, and NUM_{bc} represent the number of solutions for *block, cyclic, complete*, and *block cyclic* partitioning, respectively. We note that:

$$NUM_{cmp} = D \quad (5)$$

where D is the number of dimensions of the multi-dimensional array; that is, for each array dimension, there is a single *complete* partitioning. We also need to define a function Φ that finds the largest power-of-2 number less than the input. With reference to the variables in Table I, we have:

$$NUM_b = |B_b| = \sum_{d=0}^{D-1} log_2(\Phi(S_d)) \quad (6)$$

where S_d is the number of elements in dimension d. The solution-set size for block partitioning is equal to the cardinality of the set of allowed values for the block size (in each dimension we have the option to choose any power-of-2-sized block size). For cyclic:

$$NUM_c = |N_c| = \sum_{d=0}^{D-1} log_2(\Phi(S_d)) \quad (7)$$

Likewise, the solution-set size for cyclic partitioning is equal to the cardinality of the set of allowed values for the number of partitions. For block-cyclic partitioning:

$$NUM_{bc} = \sum_{d=0}^{D-1} \sum_{i=1}^{log_2(\Phi(S_d))-1} log_2(\Phi(S_d)) - i \quad (8)$$

The outer sum is over all dimensions. The inner sum is the solution space size for each dimension. The inner sum index i defines the legal block sizes for block cyclic partitioning: the block sizes are 2^i, and therefore range from 2^1 to $2^{log_2(\Phi(S_d))-1}$ in powers-of-2. The reason that block sizes of $2^0 = 1$ and $2^{log_2(\Phi(S_d))}$ are excluded from the sum bounds is that these solutions are standard cyclic, and block partitioning, respectively (already counted above in the NUM_c and NUM_b terms). For each legal block size, $log_2(\Phi(S_d)) - i$ counts the number of ways the block size can be assigned to banks (number of partitions).

Consider the simple case of a 2-dimensional array of 33×16 elements, D in this case is 2, S_0 is 16 and S_1 is 33. Let us take a look at the row dimension (1), $|P_b| = 5$, $|P_c| = 5$, and $|P_{bc}| = 10$. If we elaborate the block cyclic set we get $|P_{bc}| = 4 + 3 + 2 + 1$, where each term corresponds to the number of solutions for when the block size is 2, 4, 8, and 16, respectively.

The above represents the solution-set size for a single array. When there are multiple arrays within the same program, the exploration space becomes a Cartesian product of all the possible partitioning schemes of each array, which may become very large even for modestly sized arrays. The brute-forced approach for enumerating partitioning possibilities does not scale well with the size of the solution space nor with the memory trace size. In future work, a pruning approach can be used to remove partitioning possibilities when they are known to perform worse than other schemes. Large memory traces will take longer time to process, to deal with this issue, we can use a sampling approach, where smaller trace portions can be gathered at various intervals in order to get a small but representative version of the memory trace.

[1]We ignore schemes that are trivially non-optimal and that warrant no further analysis, e.g. where the number of partitions far exceeds the number of threads.

VI. Experimental Results

A. Benchmark Description

The evaluation of array partitioning on circuit performance, as well as the accuracy of the memory simulator were performed using 8 multi-threaded benchmarks: *matrixadd, histogram, matrixmult, matrixmult (cyclic), matrixtrans, matrixtrans (block cyclic), substring,* and *los*.

Matrixadd performs the summation of the elements of a 128×128 input integer matrix, where each thread is responsible for the summation of a chunk of rows. *Histogram* reads a 32768 element input array and counts the number elements which belong to a certain value range into a local array. *Matrixmult* performs matrix multiplication on two 32×32 input matrices – `matrixA` and `matrixB`, where each thread is responsible for computing the results of a chunk of rows. *Matrixmult (cyclic)* is a variation on the *matrixmult* benchmark, where each thread is responsible for computing the output for each row in a cyclic manner, where thread 0 will compute the outputs of row 0, 8, 16, etc. Additionally, *matrixmult* was modified such that each thread accesses `matrixB` with an offset to a different set of columns to allow for parallelization of accesses to columns. *Matrixtrans* performs matrix transpose on a 128×128 input matrix, where each thread transposes each row of the input for a contiguous slice of columns into the output. For example, thread 0 is responsible for transposing for all rows between columns 0 - 15 for the input matrix. *Matrixtrans (block cyclic)* is a variant on *matrixtrans* benchmark, where each thread transposes a chunk of rows of the input for all columns into the output in a block cyclic manner. For example, thread 0 is responsible for transposing rows 0-3, 32-35, 64-67, and 96-99 for all columns of the input matrix. *Los* (line of sight) consists of a 64×64 obstacle map in which each pixel can take on the values of either 1 or 0, where 1 represents an obstacle and 0 represents unoccupied space and a human is assumed to be situated in the middle of the obstacle map. The threads will sweep through the obstacle map at each map coordinate and steps in the direction of the human located at the center to determine whether there is an obstacle between the beginning location and the human. If there is, then that coordinate location is not within the line of sight of the human and thus will be marked as 0 on the output map, otherwise it will be marked 1. This benchmark is input dependent and is not intuitive in terms of which partitioning scheme is the most optimal.

B. Partitioning Performance Results

In the performance evaluation of our array partitioning pass, we run each benchmark with 8 threads across all partitioning schemes, where the number of partitions does not exceed 16. The notation used for the partitioning schemes for the rest of the paper is as follows:

$$\langle dimension\ of\ partition\rangle\langle partition\ type\rangle\langle n\rangle[\_\langle b\rangle]$$

where partition type is one of the following $*, b, c, bc$ which stands for complete, block, cyclic, or block cyclic. For example, if we wanted to partition a $M \times N$ matrix into 2 blocks, where the first block contains the elements in all the rows for columns in range 0 to $N/2-1$ and the second block contains the elements in all the rows for columns in range $N/2$ to $N-1$, then the partitioning scheme would be $0b2$ which stands for block partition into 2 blocks in the column dimension.

First, we present the execution cycles reduction results in Table II. The second column shows the cycle count for the baseline unpartitioned case for the parallel kernel region only (not including error checking); the third column shows the cycle count of the partitioned cases leading to the lowest cycle count also for the parallel kernel region; the fourth column shows the cycle of the last memory access to the arrays under partitioning, which will be useful for comparison in Section VI-D; the fifth column shows the cycle count speedup of the partitioned over the unpartitioned case; and column six shows the partitioning scheme(s) that lead to the best cycle count. In *matrixmult, matrixmult (cyclic), matrixtrans*, and *matrixtrans (block cyclic)* the benchmarks each contain 2 matrices, and both can be partitioned. The results presented show the partitioning schemes leading to the best cycle execution for the matrix `matrixA` in *matrixmult* and *matrixmult (cyclic)*, and for the matrix `input_array` in *matrixtrans* and *matrixtrans (block cyclic)*. Since the access patterns for the 2 matrices in these 4 benchmarks are transposed of each other, i.e. for *matrixmult* `matrixA` is accessed row by row while `matrixB` is accessed column by column, the partition scheme which leads to the best cycles execution in one matrix can simply be applied to the other matrix in the opposite dimension. This means, the partitioning scheme leading to one of the lowest cycle count for *matrixadd* is $0b16$ for `matrixA` and $1b16$ for `matrixB`. The geomean speedup of all 8 benchmarks is 2.21×.

From Table II we observe that generally multiple partitioning schemes may yield the same cycle count. However, we know intuitively that partitioning schemes requiring more partitions will yield designs with lower clock frequencies and more area since higher numbers of partitions means more arbitration logic. We will discuss area versus performance trade-offs in the following section.

C. Partitioning Performance vs. Area

As a representative example, we examine the *matrixadd* benchmark in detail and assess its performance versus area trade-offs.

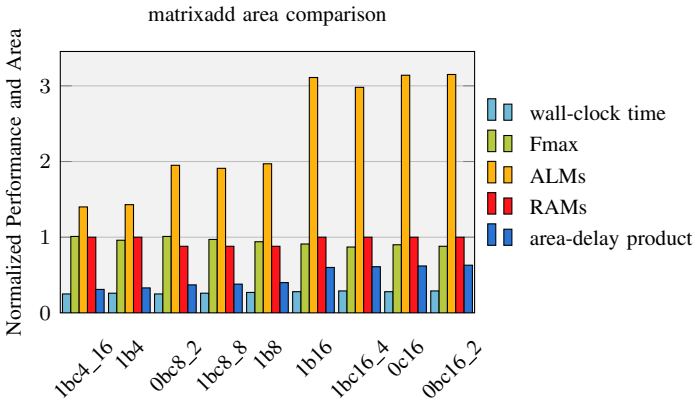
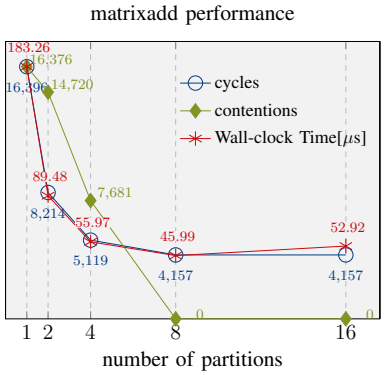


Fig. 3: Area and wall-clock time variations among "best" performing *matrixadd* partitions.

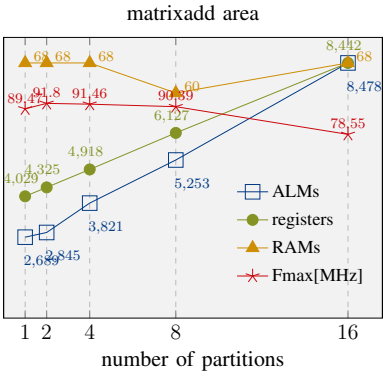
Fig. 3 shows the relative wall-clock time, Fmax, ALMs and RAM usage, and area-delay products of the 9 partitioning schemes leading to the best cycle count found through ModelSim simulation. All values are normalized to the unpartitioned design results. The area-delay product is calculated using the relative area-scaling factors presented in [14]. The scaling

benchmark	parallel region cycles		last memory cycle	speedup	partition schemes
matrixadd	16396	4157	4163	3.94	1b4, 1b8, 1b16, 0c16, 0bc8_2, 0bc16_2, 1bc4_16, 1bc8_8, 1bc16_4
histogram	33077	24013	23987	1.38	0b8, 0b16
matrixmult	32839	16799	16793	1.95	0b16, 1b4, 1b8, 1b16, 0c8, 0c16, 1c16, 0bc4_2, 0bc8_2, 1bc4_4, 1bc8_2
matrixmult (cyclic)	32837	16794	16789	1.95	0b16, 1b16, 0c8, 0c16, 1c4, 1c8, 1c16, 0bc4_2, 0bc8_2, 1bc4_2, 1bc8_2
matrixtrans	16395	6423	6419	2.55	0b4, 0b8, 0b16, 0c8, 0c16, 0bc4_2, 0bc8_2, 0bc8_4, 0bc8_8, 0bc16_2, 0bc16_4
matrixtrans (blockcyclic)	16482	6348	6344	2.60	0c8, 0c16, 1c16, 1b16, 0bc4_2, 0bc8_2, 0bc16_2, 1bc4_4, 1bc4_8, 1bc8_2, 1bc8_4, 1bc8_8, 1bc16_2, 1bc16_4
substring	16396	4120	4117	3.98	0c4, 0c8, 0c16, 0b8, 0b16, 0bc8_2, 0bc8_1024, 0bc16_2, 0bc16_512
los	83217	79543	79537	1.05	1bc16_2

TABLE II: Baseline unpartitioned versus best partition scheme(s) execution cycles.



(a) Performance result comparing total execution cycles of benchmark as well as the number of resulting contentions and wall-clock time computed using the Fmax achieved.



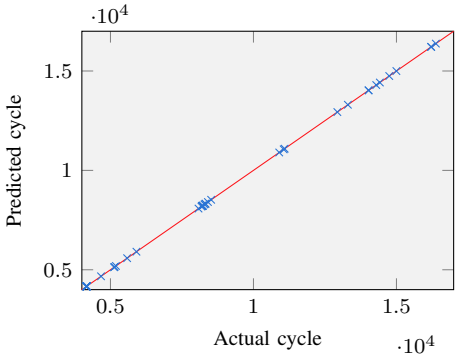
(b) Area result comparing total number of ALMs, registers, block RAMs, as well as the Fmax achieved after static timing analysis.

Fig. 4: Performance and area comparison for *matrixadd* benchmark with block cyclic partitioning across different numbers of partitions with block size 2.

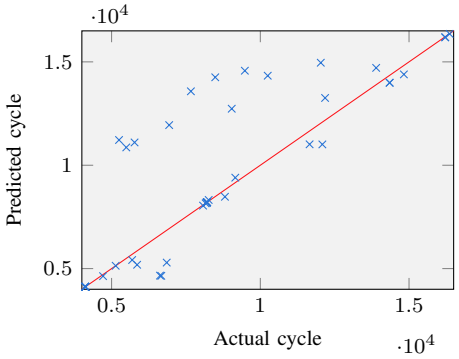
factor for the ALUT (half-ALM) is presented as 0.05 in [14], which we multiply by 2 to estimate the area of a full ALM. The scaling factor for an M9K RAM block is given as 2.87 in [14], which we multiply by 1.1 to estimate the area of an M10K RAM. Notice that even among these "optimal" partitionings, the normalized area-delay product fluctuates between 0.31 to 0.63. The main cause of this is due to the high area cost of partitioning in some of these schemes.

It is also important to observe that partitioning leads to diminishing returns in terms of cycle count, but may incur significant area costs. Fig. 4a and Fig. 4b show the performance and area of the *matrixadd* benchmark respectively under block-cyclic partitioning in the column dimension (0) for various numbers of partitions and a block size of 2. Observe that the execution cycles reaches a minimum for 8 and 16 partitions, however, the clock frequency reduces between 8 partitions to 16 partitions from 90.39 MHz to 78.55 MHz, while the ALM, register, and RAM usage all increase as we expect.

D. Evaluation of the Memory Simulator



(a) *matrixadd* benchmark



(b) *substring* benchmark

Fig. 5: Memory simulator predicted versus actual cycle of last memory access.

The goal of the memory simulator is to accurately predict the schedule of memory accesses of the circuit under various partitioning schemes from a memory access trace gathered from an unpartitioned implementation of the circuit under a representative input. The size of the exploration space is described in Section V-D. In this evaluation, we perform a prediction of the execution cycle count under all possible partition schemes where the number of partitions does not exceed 16 and compare the results to the results from Table II in Section VI-B to see if *i)* the optimal partitioning schemes were detected and if *ii)* the predicted cycle of last memory access reported by the memory simulator matches the actual cycle from ModelSim simulation.

Table III shows the predicted and actual cycle of last memory access of the partitioned arrays, the absolute percent-

benchmark	predicted cycle of last memory access	actual cycle of last memory access	% error	partitioning schemes
matrixadd	4163	4163	0	1b4, 1b8, 1b16, 0c16, 0bc8_2, 0bc16_2, 1bc4_16, 1bc8_8, 1bc16_4
histogram	23987	23987	0.004	0b8, 0b16
matrixmult	16793	16799	0.03	0b16, 1b4, 1b8, 1b16, 0c8, 0c16, 1c16, 0bc4_2, 0bc8_2, 1bc4_4, 1bc8_2
matrixmult (cyclic)	16789	16789	0.16	0b16, 1b16, 0c8, 0c16, 1c4, 1c8, 1c16, 0bc4_2, 0bc8_2, 1bc4_2, 1bc8_2
matrixtrans	6420	6419	0.01	0b4, 0b8, 0b16, 0c8, 0c16, 0bc4_2, 0bc8_2, 0bc8_4, 0bc8_8, 0bc16_2, 0bc16_4
matrixtrans (blockcyclic)	6345	6344	0.01	0c8, 0c16, 1c16, 1b16, 0bc4_2, 0bc8_2, 0bc16_2, 1bc4_4, 1bc4_8, 1bc8_2, 1bc8_4, 1bc8_8, 1bc16_2, 1bc16_4
substring	4117	4117	16.52	0c8, 0c16, 0b8, 0b16, 0bc8_2, 0bc8_1024, 0bc16_2, 0bc16_512 (missing: 0c4)
los	79537	79537	0	1bc16_2

TABLE III: Memory simulator evaluation for 16 partitions and under.

age error for all cases of partitioning, where the number of partitions is equal or less than 16, as well as the partition schemes predicted by the memory simulator to have the lowest last cycle of memory access. Fig. 5a shows the accuracy of the memory simulator for the *matrixadd* benchmark, where the horizontal axis plots the actual cycle of the last memory access to the arrays of interest and the vertical axis plots the predicted cycle of last memory access. The closer to the red line the data falls, the more accurate the prediction. Fig. 5b shows the accuracy of the memory simulator for the *substring* benchmark. The absolute percentage error for the *substring* benchmark is 16.52%. The source of error was determined to be differences in scheduling due to different port bindings for memory accesses between the original unpartitioned case, from which we extract the memory access chain, and the memory port binding during the HLS of the partitioned case.

For 7 of the 8 test cases, the memory simulator was able to accurately predict the partitioning schemes which lead to the lowest execution cycles. In the case of *substring*, the memory simulator missed 1 of the 9 best partitioning schemes. These results are encouraging since looking from the variety of best partitioning schemes, we see that many of these schemes are non-intuitive. This means that it is hard for a designer to realize that the partitioning scheme $1bc4\_16$ for the benchmark *matrixadd* leads to one of the best solutions. As seen from the area comparison in Fig. 3, this solution actually leads to the lowest normalized area-delay product. This is where the memory simulator-driven automatic memory partitioning tool becomes a useful tool in detecting multiple good solutions to prune the exploration space for the hardware/software designer.

VII. Conclusion

Memory access is often a performance bottleneck in computer hardware. In this paper, we considered memory architecture synthesis in the HLS context, specifically in the synthesis of parallel hardware from parallel multi-threaded software. We implemented flexible array partitioning as an LLVM compiler pass within the LegUp HLS framework. The array partitions are implemented as separate logical RAMs (banks) in the hardware, with arbiters inserted to manage concurrent accesses among threads. With the banked architecture synthesis, the geomean speed up of the parallel execution cycles is 2.21×, across 8 benchmarks.

As it is onerous for one to select an array partitioning and specify it using specialized compiler pragmas, we devised an automatic memory partitioning tool that uses a cycle accurate memory simulator, which, with an accurate model of memory access arbitration can predict with an average absolute percent error of 2.09% the last memory cycle of access of our 8 multi-threaded benchmarks. The simulator can be applied to automatically select the best partitioning for an array.

Future work will look at the interdependency of memory bank architecture synthesis and the memory port binding step of the HLS flow, as well as how selective multi-pumping of memories may be combined with the proposed partitioning techniques. The brute-force partitioning scheme enumeration approach can also be swapped for a smarter pruning approach in order to make programs with multiple arrays tractable in this trace-based solution.

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2), 2013.

[2] J. Choi, J. Anderson, and S. Brown. From software threads to parallel hardware in FPGA high-level synthesis. In *IEEE FPT*, pages 270–279, 2013.

[3] J. Choi, S. Brown, and J. Anderson. Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware. In *IEEE FPT*, pages 152–159, 2015.

[4] A. Cilardo and L. Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Trans. Archit. Code Optim.*, 11(4), Jan. 2015.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. A. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Trans. on CAD*, 30(4):473–491, 2011.

[6] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009.

[7] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides. Area implications of memory partitioning for high-level synthesis on FPGAs. In *FPL*, 2014.

[8] Intel Corp. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[9] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM CGO*, pages 75–88, 2004.

[10] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei. Efficient memory partitioning for parallel data access in multidimensional arrays. In *IEEE/ACM DAC*, 2015.

[11] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. on CAD*, 35(10):1591–1604, 2016.

[12] Y. Wang, P. Li, and J. Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *ACM FPGA*, pages 199–208, 2014.

[13] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *ACM/IEEE DAC*, 2013.

[14] H. Wong, V. Betz, and J. Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[15] Xilinx Inc. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration.html, 2017.

[16] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang. A new approach to automatic memory banking using trace-based address mining. In *ACM FPGA*, pages 179–188, 2017.