MEMORY PARALLELISM USING CUSTOM ARRAY MAPPING TO HETEROGENEOUS STORAGE STRUCTURES

Nastaran Baradaran and Pedro C. Diniz

University of Southern California / Information Sciences Institute
Marina del Rey, California, USA
e-mail: {nastaran, pedro}@isi.edu

ABSTRACT

Configurable architectures offer the unique opportunity of customizing the storage allocation to meet specific applications' needs. In this paper we describe a compiler approach to map the arrays of a loop-based computation to internal memories of a configurable architecture with the objective of minimizing the overall execution time. We present an algorithm that considers the data access patterns of the arrays along the critical path of the computation as well as the available storage and memory bandwidth. We demonstrate experimental results of the application of this approach for a set of kernel codes when targeting a Field-Programmable Gate-Array (FPGA). The results reveal that our algorithm outperforms naive and custom data layouts for these kernels by an average of 33% and 15% in terms of execution time, while taking into account the available hardware resources.

1. INTRODUCTION

Configurable architectures offer the unique opportunity of realizing hardware designs that are tailored to the specific data and computational patterns of a given application code. A common approach to increase the performance of an implementation on a custom computing machine is for designers to aggressively exploit instruction-level parallelism (ILP). This invariably leads to a substantial increase in the data bandwidth required for keeping the functional units busy. This approach exacerbates the importance of data organization, in particular arrays, in order to allow the implementation to exploit the available memory bandwidth.

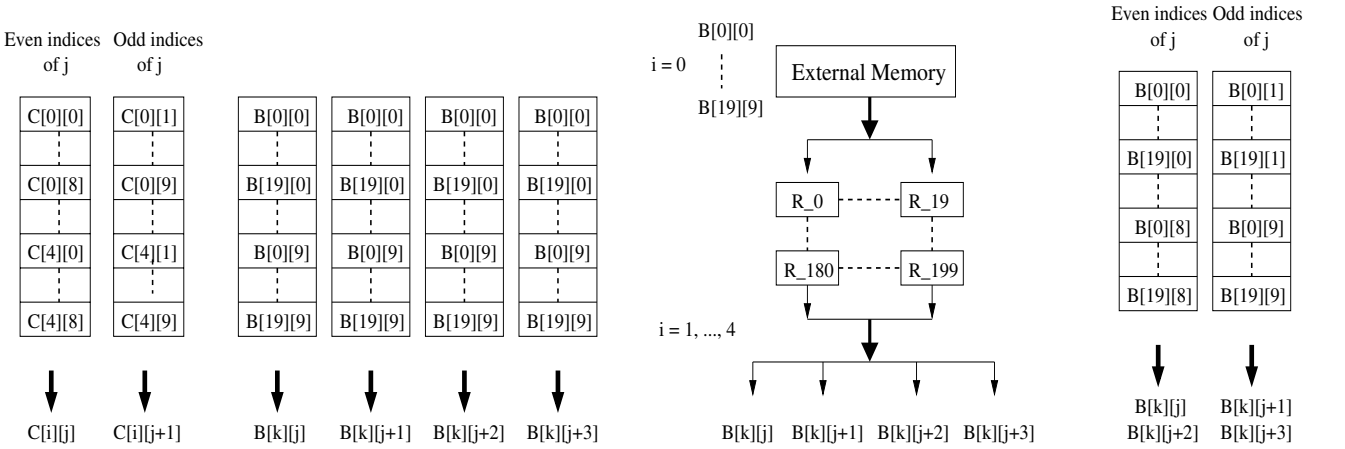
Previous compiler-based techniques for increasing the data bandwidth have focused on data distribution and neither take into account the impact of scheduling nor the computation's critical path in the application of the data-oriented transformations. On the other hand, techniques that do consider critical paths lack the high level array analysis, leading to mapping solutions that aim at reducing the execution time without any consideration of data access behavior.

In this paper we present a compiler approach and an algorithm to map a computation's data arrays to a set of heterogeneous storage resources. We consider three common data-oriented transformations, namely *distribution*, *replication*, and *scalar replacement*, and take advantage of off-chip memory, internal memory banks (RAM blocks in case of an FPGA), and discrete registers. The proposed algorithm combines the data access pattern information of the arrays with the analysis of the critical path of the computation. It applies the various data-oriented transformations to each array with the objective of minimizing the execution time subject to the available storage and bandwidth. The novelty of our approach lies in creating a single framework that combines various high-level compiler techniques with lower-level scheduling information, while considering the target architecture's resource constraints. Our preliminary results show that our algorithm leads to designs that are on average 15% and 33% faster than the designs using *custom data layout* and *naive* mapping techniques.

Our algorithm is independent of any code transformation, however it shows significant advantage in cases of loop unrolling where the execution requires a higher data bandwidth. The work here is geared towards image/signal processing computations that can be structured as perfectly or quasi-perfectly nested loops, with symbolically constant loop bounds that manipulate array variables.

In emerging multi-core architectures with a rich set of storage structures the placement and management of data will become increasingly important. Mapping algorithms based on data behavior and scheduling information, such as the one described here, will ultimately allow designers to develop better design solutions and/or explore a much wider range of design choices without becoming involved in the myriad of low-level and error-prone details.

This paper is organized as follows. In section 2 we present a motivation and background for array mapping. In section 3 we describe the compiler analyses that support the proposed array mapping algorithm. In section 4 we present preliminary experimental results of the application of our algorithm to a set of kernel codes targeting an FPGA device. We review related work in section 5 and conclude in section 6.



(a) Distribution for array C (b) Replication for array B (c) Scalar replacement for array B (d) Distribution for array B considering the scheduling

Fig. 1. Various mapping techniques applied to the example code in figure 2.

2. MOTIVATION & BACKGROUND

We now illustrate the application of the data transformations used in our mapping algorithm to an example code that manipulates array variables as depicted in figure 2.

```
for (i = 0; i < 5; i++)
 for (j = 0; j < 10; j++)
  for (k = 0; k < 20; k++){
   C[i][j] += (A[i][k] * B[k][j])/4 + B[k][j+2];
  }

for (i = 0; i < 5; i++)
 for (j = 0; j < 10; j+=2)
  for (k = 0; k < 20; k++){
   C[i][j] += (A[i][k] * B[k][j])/4 + B[k][j+2];
   C[i][j+1] += (A[i][k] * B[k][j+1])/4 + B[k][j+3];
  }
```

Fig. 2. Example code. Original (top); After unrolling the j loop by a factor of 2 (bottom).

Data Distribution: This transformation partitions the array's data into disjoint sets and maps them to distinct memory modules, thereby increasing the available bandwidth, while preserving the total storage used for holding the data. In figure 2, references C[i][j] and C[i][j+1] access non overlapping sections of the array. We can distribute the data for array C between two different memory modules, binding each reference to one memory, thus allowing concurrent accesses to the corresponding data elements (figure 1(a)).

Data Replication: This transformation increases the availability of the data by creating copies of the data in distinct memory modules. Though costly in terms of storage, it can be used profitably when the replicated data is small and frequently read, as there is an issue of consistency of replicas in the presence of write operations. In figure 2, we can replicate the B array among 4 memories and concurrently access the data referenced by B[k][j], B[k][j+1], B[k][j+2] and B[k][j+3] (figure 1(b)).

Scalar Replacement: This transformation converts array references to scalar variables and then maps them to registers. The first access to each scalar replaced data item (typically a read operation) requires a memory access, but subsequent accesses can use the data cached in the registers. This transformation, however, may be infeasible due to the large number of required registers. For example, in figure 1(c), array B would require $b_k \times b_j = 200$ registers to cache the data accessed in the first iteration of the i to be reused in the remaining iterations of the same loop.

Clearly, applying these transformations indiscriminately for all the data references may lead to capacity issues. Instead, we make the observation that the storage resources and the corresponding mapping transformations should aim at reducing the *critical path* of the computation. For example in figure 2, considering the dependency and therefore schedule, computation needs concurrent accesses only to B[k][j] and B[k][j+1]. References B[k][j+2] and B[k][j+3] are only accessed at a later time and can share the bandwidth with B[k][j] and B[k][j+1] (figure 1(d)).

The algorithm presented in this paper targets the references that have affine subscripts and are uniformly generated (see *e.g.,* [9]). In our architectural model a processor can simultaneously access various multi-ported memory banks but we are not concerned with the low-level physical memory organization of each memory bank. In our critical path analysis we assume sufficient functional units so that the memory ports are the only source of data access contention. Finally, our data mapping is fixed for the entire duration of the computation.

3. CUSTOM ARRAY MAPPING

We now describe our array data mapping algorithm that takes into account the computation's critical path, the array references' access patterns, and the available hardware resources.

Analysis of Critical Paths: Our algorithm captures the computation of the body of a loop nest as a Data-Flow-Graph (DFG) derived from the static-single-assignment (SSA) intermediate form. In the DFG edges represent data dependences and nodes (augmented with latency information) represent data accesses or arithmetic/logic operations. Control flow is represented by the graphs corresponding to the two branches of a conditional statement[1].

Given a DFG we define the *critical path* (CP) as the longest execution path(s) given the access delays of data references. A *critical graph* (CG) is a subgraph of the DFG that only includes its critical path(s). We also define a *Cut* of the CG as a *minimal* subset of its data reference nodes, such that their removal would bisect all the paths of the CG. Removing the access latencies associated with the references of a cut reduces the CPs of the DFG by a *memory latency* time [6]. The nodes of a cut need to be considered inclusively, as improving only a subset of them will not reduce the execution time of the critical graph. Figure 3 depicts the DFG and the initial CG and its cuts for the computation in the body of the example code in figure 2. *Source* and *sink* are artificially added nodes, representing the beginning and end of the computation. The four sets of cuts for this example include {B[k][j], A[i][k], B[k][j+1]}, {B[k][j], A[i][k], C[i][j+1]}, {B[k][j+1], A[i][k], C[i][j]}, and {C[i][j], C[i][j+1]}.

Analysis of Data Access Patterns: We analyze the data access patterns for the N references to each array A in order to uncover opportunities for concurrent data accesses [2].

If the references access array data sections that do not overlap, the data can be partitioned to N distinct sets. Alternatively we could apply scalar replacement to array A and simultaneously access the values corresponding to the N references directly from registers. If however the data accessed by the array references do overlap, we are required to replicate the array's data across N memories and/or apply scalar replacement to increase the availability of the data.

When there is simply not enough available bandwidth to satisfy the bandwidth requirements of a set of references, the algorithm still uses data distribution and data replication among the *available* resources. In this scenario the data references share the available memory bandwidth incurring a latency delay factor of $\lceil \frac{RequiredBandwidth}{AssignedBandwidth} \rceil$ for each array.

[1]We assume a speculative execution of all the memory operations regardless of the outcome of the control flow predicates.

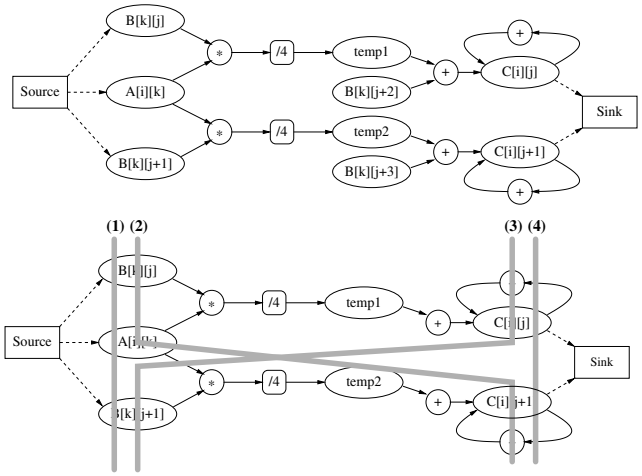


Fig. 3. DFG (top) and CG with its possible cuts (bottom).

3.1. Problem Formulation

For the array references of a given cut of the critical graph the algorithm must find a mapping strategy subject to the available number of internal registers and memory banks, such that the execution time of the cut and therefore of the critical path is minimal. In case of identical designs in terms of execution time, we select the one with the smallest size.

Here AB, RB_A, and NB_A respectively represent the total available bandwidth, the required bandwidth and the assigned bandwidth for array A. At each stage the algorithm must assign between 0 and RB_A memory bandwidth to each array variable A in a cut. If there are M arrays in the cut, the problem is defined as finding the value of NB_i ($1 \leq i \leq M$), in order to minimize $(\max_{i=1}^{M} \lceil \frac{RB_i}{NB_i} \rceil)$ subject to:

$$\begin{cases} NB_1 + \ldots + NB_M \leq AB \\ 0 \leq NB_i \leq RB_i & \text{where } 1 \leq i \leq M \end{cases}$$

This formulation naturally leads to a simple greedy (and therefore locally optimal) algorithmic solution where a single cut is examined at each algorithm step. As the result of this lack of a global perspective we expect our algorithm to derive mapping solutions that are non-optimal, but nevertheless lead to good hardware designs.

3.2. Custom Array Mapping (CAM) Algorithm

This algorithm attempts to reduce the latency of the critical path by greedily applying *scalar replacement*, *data distribution* and *data replication* to the arrays referenced by the nodes of a cut, one cut at a time.

Initially all arrays are stored in off-chip memory. The algorithm first creates the DFG and extracts the CPs and its cuts. In a selected cut [2], the array references need to be accessed in parallel with the lowest latency. As such the algo-

[2]Selecting the "best" cut can be based on different metrics and is outside the scope of this paper.

rithm first attempts to keep the data in registers by applying scalar replacement to all array references of the cut. In case of insufficient registers, the algorithm determines if the references access disjoint/overlapping sections of the arrays' data to decide whether to distribute or replicate data across RB banks. In this phase the algorithm attempts to use as many memory banks as possible to satisfy the bandwidth requirements of all array references of the cut. In case of insufficient memory banks to accommodate all the references in a cut, the algorithm uses a combination of the available registers and memory modules to minimize the memory latency of the cut. To accomplish this the algorithm exhaustively searches the possible combinations of mappings based on the access patterns and available storage (lines 13-26). The net result of this phase of the algorithm is a reduction by at most one memory latency in the critical path, depending on the mapping decisions in the selected cut.

In subsequent iterations of the algorithm, the critical path is updated to reflect the newly allocated storage and a new set of cuts is identified. After selecting the best cut, for the arrays that have not been mapped before, the algorithm searches the best mapping policy as in an earlier phase (lines 54-56). Otherwise, and for a reference r corresponding to an already mapped array A, the algorithm does not alter the current mapping policies and only updates them as follows:

Scalar Replaced: If A is already scalar replaced then r can use the data already in registers. No update in the mapping is required.

Replicated: If the current $RB_A < NB_A$, then r can simply point to the existing copies. Otherwise, the algorithm creates $RB_A - NB_A$ new copies of the array (lines 30-39).

Distributed: If r's data is a subset of an existing partition, then r will be bound to that partition. Otherwise, the algorithm needs a new bank to accommodate the new partition (lines 40-49).

The algorithm continues by updating the critical path and removing/improving the memory latencies until it exhausts all available storage. The exhaustive search can make the algorithm exponential in the worst case, however in practice, and due to the limited number of arrays in a cut, the search space is small enough for a brute-force approach.

Here we apply the above algorithm to the example code in figure 2 with its DFG and CG depicted in figure 3. We assume 4 memory banks, M0, ... , M3 and select the best cuts based on the minimum amount of required storage.

For the selected cut labeled as (4), since there is no data reuse for C[i][j] and C[i][j+1] and as they access disjoint data, the algorithm distributes the array across M0 and M1. During the second phase the updated critical path leads to only one possible cut of {B[k][j], A[i][k], B[k][j+1]}. A fully parallel data access requires 3 banks. Due to insufficient number of available banks, the algorithm maps one array, A, to registers and the other array to the available 2

```
Input:  Data Flow Graph of a normalized loop.
Output:  Mapping of each array.

 MappingAlgorithm
1. firstCut = true;
2. for each array A do {
3.  NB_A = 0;Delay_A = 0;
4. }
5. while there is more storage do{
6.  Make the Critical Graph;
7.  Extract the set of Cuts;
8.  For a selected cut C {
9.   Delay_C = 0;
10.  for each array A in the C do
11.   Analyze and calculate RB_A;
12.  // First iteration
13.  if (firstCut) then{
14.   if(enough registers) then
15.    scalar replace all the arrays;
16.    Delay_A = Lat_Reg;
17.   elseif(RB_A < AvailableRAMs)
18.    Distribute/Replicate all the arrays;
19.    Delay_A = Lat_RAM;
20.    NB_A = RB_A; //for all A ∈ C
21.   else
22.    for all the possible mappings do {
23.     find the mapping that minimizes Delay_C;
24.     Update NB_A; //for all A ∈ C
25.     Update Delay_A; //for all A ∈ C}
26.   Update the available storage;
27.   } else // later iterations
28.   // Update the mapped arrays
29.   for each array A of the cut C do
30.    if (previously replicated) then {
31.     if (RB_A ≤ NB_A) then
32.      Just point to the same RAMs;
33.     else
34.      if (enough storage) then
35.       Create another RB_A − NB_A copies of A;
36.       Update NB_A;
37.      else
38.       Assign references to the available RAMs;
39.    }
40.    elseif (previously distributed) then
41.     if (references point to the existing data)
42.      point to the corresponding RAMs;
43.     else
44.      if (enough storage) then
45.       create the new partitions;
46.       Update NB_A;
47.      else
48.       Access external memory
49.    }
50.    Update Delay_A;
51.    Delay_C = Max(Delay_C, Delay_A)
52.   }// end for
53.   // create new mappings for unmapped arrays
54.   for each non mapped array X in C do {
55.    find NB_X such that ⌈RB_X / NB_X⌉ ≤ Delay_C;
56.    Update NB_X;}
57.   Update the available storage
58.  }
59. }
60.}// end while
```

Fig. 4. Custom Array Mapping Algorithm.

Table 1. Experimental Results.

Kernel	Version	Naive				CDL				CAM					ET Gain(%)		Cycle Gain(%)	
		Cycles	Clk(ns)	ET(ms)	Slices	Cycles	Clk(ns)	ET(ms)	Slices	Cycles	Clk(ns)	ET(ms)	Slices	Reg(bits)	over Naive	over CDL	over Naive	over CDL
FIR	11	119810	25.96	3.1	311	102402	23.01	2.35	233	102402	23.01	2.35	233	0	24.24	0	14.52	0
	12	78850	26.15	2.1	377	61442	23.51	1.44	294	56322	22.95	1.29	304	0	37.31	10.51	28.57	8.33
	21	69634	28	1.95	421	59906	22.03	1.32	250	52738	25.98	1.37	359	0	29.72	-3.81	24.26	11.96
	14	58370	27.8	1.62	464	40962	26.91	1.10	441	35842	29.46	1.05	477	0	34.92	4.21	38.59	12.50
	41	44546	28.46	1.26	628	38914	27.29	1.06	525	26671	28.55	0.76	787	228	39.94	28.29	40.12	31.46
	22	49154	28.84	1.41	571	40450	29.26	1.18	462	28207	27.72	0.78	666	204	44.84	33.93	42.61	30.26
PAT	11	109058	23.66	2.58	221	100354	19.3	1.93	137	100354	19.3	1.93	137	0	24.93	0	7.98	0
	12	64002	26.37	1.68	266	55298	19.9	1.10	195	52738	22.41	1.18	204	0	29.97	-7.39	17.59	4.62
	21	59394	26.08	1.54	319	54530	23.05	1.25	262	50946	25.2	1.28	245	0	17.11	-2.14	14.22	6.57
	14	41474	26.54	1.10	323	32770	24.64	0.80	228	30210	25.99	0.78	234	0	28.66	2.76	27.15	7.81
	41	34562	25.93	0.89	410	31746	28.53	0.90	342	25647	28.65	0.73	606	152	18.01	18.87	25.79	19.21
	22	34818	26.53	0.92	351	30466	24.9	0.75	265	26415	26.37	0.69	287	136	24.59	8.17	24.13	13.29
JAC	11	9060	21.68	0.19	254	9060	20.6	0.18	233	9060	20.6	0.18	233	0	4.98	0	0	0
	12	7710	28.04	0.21	374	6360	27.33	0.17	325	3660	30.05	0.11	1078	576	49.12	36.72	52.52	42.45
	21	7680	28.4	0.21	319	6330	27.02	0.17	349	4530	21.57	0.09	341	0	55.20	42.87	41.01	28.43
	14	6750	30.97	0.20	533	4620	29.95	0.14	544	2610	40.13	0.10	2400	576	49.89	24.30	61.33	43.50
	41	6708	30.21	0.20	596	4818	30.11	0.14	575	3438	29.49	0.10	602	0	49.96	30.11	48.74	28.64
	22	6105	29.35	0.17	478	4080	28.64	0.12	391	2955	27.61	0.08	417	0	54.46	30.17	51.59	27.57
BIC	11	107708	23.48	2.52	295	107708	21.09	2.27	210	67355	29.99	2.01	561	128	20.12	11.07	37.45	37.46
	12	67340	25.64	1.72	318	67340	24.97	1.68	239	41371	30.05	1.24	702	128	27.99	26.06	38.56	38.56
	21	62294	29.37	1.82	319	67340	22.39	1.50	243	36499	30.15	1.10	737	128	39.85	27.01	41.40	45.79
	14	43792	29.06	1.27	328	37064	24.02	0.89	259	25131	35.01	0.87	705	128	30.86	1.17	42.61	32.19
	41	43792	28.07	1.22	316	37064	23.84	0.88	258	25131	30.06	0.75	693	128	38.54	14.50	42.61	32.19
	22	45474	26.7	1.21	372	42110	26.07	1.09	324	26755	30.18	0.80	864	128	33.49	26.44	41.16	36.46

banks. The alternative choice of mapping A to the banks and B to registers would require 200 registers and therefore a larger design area. During the final phase the updated critical path information leads to only one possible cut {B[k][j+2], B[k][j+3]}. Since there has been a previous mapping decision for B, and the new references' data are a subset of the data in the previous mapping, the new references can point to the data previously mapped to M2 and M3. In the final mapping array C is distributed across M0 and M1, array B is distributed across M2 and M3, and array A is scalar replaced in registers. The execution time is shortened by two memory access delays.

4. EXPERIMENTS

We validated the CAM algorithm outlined in section 3.2 for a set of four image and signal processing code kernels. The `FIR` code kernel computes the convolution of a vector's values against a sequence of coefficients. The `PAT` kernel finds the various occurrences of a character pattern in a string, while the `JAC` kernel performs a Jacobi stencil relaxation computation over a 2-D array variable. Finally, `BIC` computes a Binary-Image-Correlation between a template image and successively overlapping regions of a larger image. These kernels are respectively structured as 2, 2, 2, and 4-deep loop nests with compile-time known loop bounds.

For each kernel, written in C, we applied loop unrolling and the various *array mapping* transformations at the source C level. We then converted the transformed C codes to behavioral VHDL and into structural VHDL designs using Mentor Graphics' Monet™ high-level synthesis tool. We used Synplify Pro 6.2 and Xilinx ISE 4.1i toos for logic synthesis and Place-and-Route (P&R) targeting a Xilinx Virtex™ XCV 1K BG560 device. We calculated the wall-clock execution time based on the number of cycles derived from the simulation and the clock rate extracted from the P&R phase.

In these experiments we imposed a maximum of 64 registers to store the arrays data. In practice this limit is set by the compiler as part of a global resource allocation policy, orthogonal to these experiments. We further assumed only 4 single-ported memory banks (RAM blocks) available. External memory accesses take 5 cycles while accessing the RAMs take only 2 cycles. All the memory accesses are fully pipelined with an initiation interval of 1 clock cycle.

For each code kernel we derived 18 designs, reflecting different mapping strategies as well as various unroll factors. In the *Naive* mapping all the arrays are mapped to the same RAM, regardless of the access pattern or the schedule. In the *Custom Data Layout (CDL)* [2] arrays are distributed among different RAMs based on their access pattern, however, irrespective of the schedule. In the CAM mapping arrays are mapped using our algorithm thus taking into account both the access pattern and how they are scheduled.

Table 1 depicts the timing and area results for the various designs. In column 2 the value of 'XY' denotes the loop unrolling factors of the two innermost loops. For each mapping strategy, *i.e.,* Naive, CDL, and CAM, the respective columns indicate the number of cycles reported by the synthesis tool, clock period reported by the P&R tool, calculated wall clock time, number of slices and number of register bits used. Finally, columns 16-19 reflect the amount of improvement for the CAM algorithm over the Naive and CDL mappings, in terms of execution time and number of cycles.

As expected and illustrated by the results, considering

the scheduling leads to a better allocation of resources and hence to a reduction in the number of execution cycles. The CAM designs have a clear advantage over their corresponding Naive and CDL designs in *all* cases, respectively gaining 33.5% and 22.5% fewer cycles on average, in some cases not even taking advantage of discrete registers. For target configurable architectures where the clock rate is fixed regardless of the design complexity, the results would reveal performance improvements for all code variants as derived from the reduction of the number of clock cycles.

With respect to the clock period, CAM versions suffer from a 10% average clock degradation compared to CDL designs. This is mainly due to the complexity of the algorithm as well as the use of scalar replacement that increases the design size. The CAM algorithm however decreases the number of clock cycles enough to compensate for the degradation in the clock rate, gaining an average 33.7% and 15.2% speedup over Naive and CDL designs.

In terms of area, and for cases with no scalar replacement, the performance of CAM is comparable to Naive and CDL. For the code versions that include some scalar replacement the consumed area in CAM increases due to the use of registers. However, one must note that we consider the available storage (area) as an input constraint and find the fastest design that fits in this area. Therefore we can easily select a smaller design by changing these constraints if necessary.

In general the performance gains are higher for cases with a large number of overlapping references, for instance created by loop unrolling, as CDL maps these references to the same RAM block. Overall, custom array mapping improves the performance by better utilization of the *available* resources, making it an effective mapping algorithm for this class of configurable computing architectures.

5. RELATED WORK

Proper data mapping is of major importance in increasing the bandwidth and hence reducing the overall execution time. The earlier work in [1] describes an algorithm based on a precedence graph without considering the data access patterns. In contrast the approaches in [2, 5] map the distinct sections of an array to different RAMs based on the data access patterns of array references and regardless of the schedule. Various authors (*e.g.* [6, 8]) use the scheduling and data reuse information in order to exclusively map the arrays to RAM blocks or registers. The work in [3, 11] identifies the footprint of each array to allocate array variables to memories, while the approach in [4] describes a methodology to cache the reusable data in the smaller RAMs. Another body of work (*e.g.* [7, 10]) focuses on customizing the local memory based on the application's characteristics.

The work described here differs from these efforts in that it uses a precise notion of access pattern/reuse analysis, exploits the notion of precedence by focusing on the critical paths of the computation, and finally uses several compiler transformations and different storage resources.

6. CONCLUSION

In this paper we described a compiler data mapping algorithm that uses the data access pattern information of array variables to map them to available storage resources. The algorithm applies a set of data-oriented transformations and exploits the available bandwidth guided by the critical path information in the computation. The preliminary results, for a set of image/signal processing kernels targeting a Xilinx FPGA device, exhibit an average 33% and 15% speedup over the *naive* and the recently proposed *custom data layout* mapping approaches.

7. REFERENCES

[1] M. Gokhale, J. Stone. Automatic Allocation of Arrays to Memories in FPGA Processor s with Multiple Memory Banks. *IEEE Symp. on FPGAs for Custom Computing Machines*, 1999.

[2] B. So, M. Hall, H. Ziegler. Custom Data Layout for Memory Parallelism. *Intl. Symp. on Code Gen. and Opt.*, 2004.

[3] W. Gong, G. Wang, and R. Kastner. Storage Assignment during High-Level Synthesis for Configurable Architectures. *ACM/IEEE Intl. Conf. on Computer-Aided Design*, 2005.

[4] F. Balasa, F. Catthoor and H. de Man. Practical Solutions for Counting Scalars and Dependences in ATOMIUM - A Memory Management System for Multidimensional Signal Processing. *iEEE Trans. on Computer-Aided Design and Integration of Circuits and Systems*, 16(2):133–145,1997.

[5] R. Barua, W. Lee, S. Amarasinghe and A. Agarwal. Maps: A Compiler-Managed Memory System for RAW Machines. *Intl. Symp. on Computer Architecture*, 1999.

[6] N. Baradaran and P. Diniz. A Register Allocation Algorithm in the Presence of Scalar Replacement for Fine-Grain architecture. *Design Automation and Testing in Europe*, 2005.

[7] M. Kandemir, A. Choudhary. Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. *Proc. ACM/IEEE Design Automation Conf.*, 2002

[8] M. Weinhardt, W. Luk. Memory Access Optimization for Reconfigurable Systems. *IEE Proc.-Comput. Digit. Tech.*, 148(3), pp. 105–112, 2001.

[9] M. Wolf and M. Lam. A Data Locality Optimization Algorithm. *In Proc. of the ACM Conf. on Programming Language Design and Implementation*, pp. 30-44, 1991.

[10] P. Jha and N. Dutt. High-level Library Mapping for Memories. *ACM Trans. on Design Automation of Electronic Systems*, 5(3):566–603, January 1999.

[11] D. Bairagi, S. Pande and D. Agrawal. Framework for Containing Code Size in Limited Register Set Embedded Processors. *ACM Workshop on Languages, Compilers and Tools for Embedded Systems*, 2000.