Custom Data Layout for Memory Parallelism

Byoungro So
IBM T. J. Watson Research Center
Exploratory System Architecture
1101 Kitchawan Road / Route 134
Yorktown Heights, NY 10598
bso@us.ibm.com

Mary W. Hall and Heidi E. Ziegler
University of Southern California
Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, CA 90292
{mhall,ziegler}@isi.edu

Abstract

In this paper, we describe a generalized approach to deriving a custom data layout in multiple memory banks for array-based computations, to facilitate high-bandwidth parallel memory accesses in modern architectures where multiple memory banks can simultaneously feed one or more functional units. We do not use a fixed data layout, but rather select application-specific layouts according to access patterns in the code. A unique feature of this approach is its flexibility in the presence of code reordering transformations, such as the loop nest transformations commonly applied to array-based computations. We have implemented this algorithm in the DEFACTO system, a design environment for automatically mapping C programs to hardware implementations for FPGA-based systems. We present experimental results for five multimedia kernels that demonstrate the benefits of this approach. Our results show that custom data layout yields results as good as, or better than, naive or fixed cyclic layouts, and is significantly better for certain access patterns and in the presence of code reordering transformations. When used in conjunction with unrolling loops in a nest to expose instruction-level parallelism, we observe greater than a 75% reduction in the number of memory access cycles and speedups ranging from 3.96 to 46.7 for 8 memories, as compared to using a single memory with no unrolling.

1. Introduction

The performance gap between processor and memory speeds [23] has inspired numerous strategies over the last several years to increase memory bandwidth. For example, there is a recent trend toward tightly coupling multiple independent memories or memory banks with one or more functional units, so that multiple independent memory accesses occur in parallel. This approach is being used at all system scales and for a variety of architectural platforms, from Blue Gene L at the high end, to a number of recent systems-on-a-chip research platforms, including MIT's Raw [3], UT's TRIPS [1] and Stanford's Smart Memories [13], as well as a growing number of commercially available embedded processors, such as Motorola DSP56000 series, NEC 77016, SONY pDSP, Analog Devices ADSP-210x, Starcore SC140 processor core, and FPGA-based computing platforms from Annapolis Systems and others.

¿From a software perspective, these architectures increasingly expose data storage and data movement to software control, enabling intelligent compilers or heroic application developers to simultaneously organize data and computation so that the full memory bandwidth of the architecture is exploited.

There are a number of techniques being developed to increase the likelihood that nearby memory accesses will be to independent memories, thus enabling parallel memory accesses. Most compiler solutions assume a fixed data layout in memory, usually standard row or column major, and transform the code to access memory in parallel whenever possible. A number of techniques examine the low-level output of the compiler and, given a fixed mapping of data to memory, reorder individual accesses and operations to increase memory parallelism [24]. Most closely related to our work is what is done for array-based computations in the Raw compiler [4], where data layouts are in *modulo unrolling* order, a fixed layout, cyclic across multiple memory banks in the most quickly varying dimension of the array. The Raw compiler combines loop unrolling with renaming accesses and offsets to derive optimized code.

In this paper, we propose an alternative approach whereby we derive a *custom data layout* in memory in an application-specific way, according to the access patterns in a loop nest computation. Our approach is more flexible than techniques that assume a fixed data layout. This difference is particularly important when used in conjunction with code reordering transformations, such as loop nest transformations commonly performed on array-based computations. Our compiler has more degrees of freedom in transforming code, and can thus preserve memory parallelism while accomplishing other optimization goals. For data structures with different access patterns in different portions of the code, the compiler can choose between reducing memory parallelism within each computation, or reorganizing the data between computations, depending on what is more profitable.

Our approach has been fully automated in the DEFACTO compiler [20], a Design Environment For Adaptive Computing TechnOlogy, which integrates compilation and behavioral synthesis to automatically map computations expressed in C to field programmable gate array (FPGA) based computing platforms. The target architectures do not have data or instruction cache, but have multiple external memories feeding a single FPGA in parallel; internal to the FPGA, significant instruction-level parallelism is exploited on the computation. Thus, although this is a novel computing platform, it nevertheless shares commonalities with more conventional ones in which either multiple memories feed a single or multiple functional units in parallel.

A previous paper motivated the need for this approach, and presented a high level overview of the algorithm [21]. In this paper, we present a detailed algorithm that derives a custom mapping in memory directly from the subscript expressions, so that the array is mapped to memory according to access patterns in the code. We also describe how to reorganize the data in memory from the standard layout in a single memory to the custom layout in multiple memories. We provide proofs of correctness of the approach. We present a comprehensive set of performance results, derived automatically by our compiler for five multimedia kernels.

The organization of the paper is the following. The next section describes an example to motivate the approach. Section 3 presents the the overview of the *custom data layout* algorithm. Section 4 describes the analyses and transformations to identify the parallel memory accesses in virtual memories and how we reorganize array data from/to a naïve layout in a single memory to/from a custom layout in multiple memories. Section 5 describes how to map virtual memories to a limited number of physical memories. Section 6 presents a set of experimental results derived automatically by our compiler. We survey related work in Section 7. In Section 8 we present some preliminary conclusions and future work.

2. Motivation

We illustrate the difference among several different data layouts using the example code in Figure 1(a). At the compiler level, the standard approach is to view data as being mapped to a single memory. We refer to this as a naïve data layout. Figure 1(b) shows the code after we unroll loops *i* and *j* by one (equivalent to an unroll factor of two) and jam the copies of the loop *j*. Figure 2(a) depicts a possible naïve layout for the transformed code shown in Figure 1(b). The entire array *A* and the entire array *B* are mapped into memory *M0*. Even though the code is transformed to execute four loop iterations in parallel, it will execute serially due to memory stalls. Additionally, other memories will be idle while the computation stalls for data.

```
int A[32][16];   int B[32][16];
for (i = 0; i < 32; i++)
  for (j = 0; j < 16; j++)
    A[i][j] = B[i][j] + 1;
```

(a) Original code

```
for (i = 0; i < 32; i+=2)
  for (j = 0; j < 16; j+=2) {
    A[i][j] = B[i][j] + 1;
    A[i][j+1] = B[i][j+1] + 1;
    A[i+1][j] = B[i+1][j] + 1;
    A[i+1][j+1] = B[i+1][j+1] + 1;
  }
```

(b) After unroll-and-jam

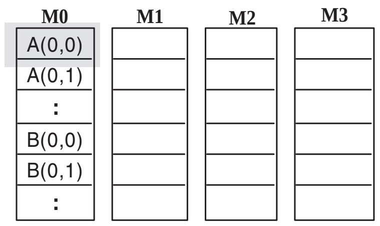
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 8; j++) {
    A00[i][j] = B00[i][j] + 1;
    A01[i][j] = B01[i][j] + 1;
    A10[i][j] = B10[i][j] + 1;
    A11[i][j] = B11[i][j] + 1;
  }
```

(c) Final code

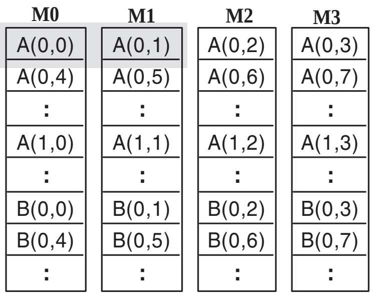
Figure 1. An example of custom data layout.

Modulo unrolling involves unrolling a loop in the nest that accesses the lowest order dimension of an array so that the accesses in an unrolled loop body are to statically fixed memory banks. Figure 2(b) shows the modulo unrolling layout for the code in Figure 1(b). This layout scheme distributes across four memories in a cyclic fashion the elements of array *A* and *B* along the second array dimension, which is the fastest changing dimension in the loop nest. The code in Figure 1(b) now can fetch two elements of array *B* at a time. Two memory banks are still idle while the computation stalls for the other two array elements. It is only if the *j* loop were unrolled by four, that modulo unrolling would fetch all four elements of *B* at the same time and thus take advantage of the available parallelism in the system.

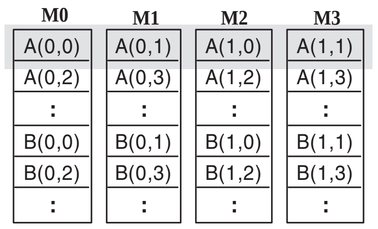
Figure 2(c) illustrates the custom data layout for the code in Figure 1(b). Custom data layout distributes arrays *A* and *B* across four memories according to the particular data access pattern in the kernel. As such, we reduce the possibility of computation stall due to memory accesses. It distributes



(a) Naïve data layout.


M0
M1
M2
M3
A(0,0) A(0,1) A(0,2) A(0,3)
A(0,4) A(0,5) A(0,6) A(0,7)
A(1,0) A(1,1) A(1,2) A(1,3)
B(0,0) B(0,1) B(0,2) B(0,3)
B(0,4) B(0,5) B(0,6) B(0,7)

(b) Modulo unrolling.


M0
M1
M2
M3
A(0,0) A(0,1) A(1,0) A(1,1)
A(0,2) A(0,3) A(1,2) A(1,3)
B(0,0) B(0,1) B(1,0) B(1,1)
B(0,2) B(0,3) B(1,2) B(1,3)

(c) Custom data layout.

Figure 2. Comparison of three data layouts.

the four array elements, $B[i][j]$, $B[i][j{+}1]$, $B[i{+}1][j]$, and $B[i{+}1][j{+}1]$, accessed in the unrolled loop body. Since all four memory elements required by the loop body are placed in separate memory banks, no memory will be idle, thus achieving better use of the machine's memory to computation bandwidth. The four statements in the loop body can now execute in parallel since there are no data dependences among them and each of the array references is accessing a different memory bank.

This example illustrates an advantage of using a custom data layout as opposed to a fixed layout such as the cyclic one used by modulo unrolling. A compiler might want to apply iteration-space reordering and code reordering transformations that conflict with the modulo unrolling transformation, thus impacting memory parallelism. For example, the DEFACTO compiler uses unroll-and-jam to increase both memory parallelism and instruction-level parallelism. Unrolling outer loops provides additional opportunities for memory parallelism which can not be exploited by modulo unrolling. Further a custom layout gives the compiler more freedom to accomplish other optimization goals without impacting memory parallelism.

1. Normalize the step size of each loop to have a unit stride.
2. *Virtual mapping* for each array; *i.e.*,
   (a) Analyze the data access pattern from the subscript expression in each array dimension of each array reference; *i.e.*, a stride and an offset.
   (b) Partition array references such that accesses to independent array elements belong to separate partitions. In each partition, we reanalyze the data access patterns of references.
   (c) Map each partition to a separate virtual memory, and rewrite array references to represent the accesses to different virtual memories by combining the results of rewriting each dimension.
   (d) Where needed, insert code to copy data to/from naïve layout in one memory from/to a custom data layout across multiple memories.
3. *Physical mapping*: based on the access orders of the arrays accessed in a particular loop nest, and taking memory operation scheduling scheme into account, bind virtual memories to physical memories.

Figure 3. Steps of custom data layout.

3. Overview

Custom data layout increases the parallel array accesses across multiple memories using a three-phase algorithm, independent of other common loop transformations. Figure 3 provides an outline of the custom data layout algorithm. The first step is normalizing the step size, which involves replacing all the instances of the loop index variable l with $s \times l$, where s is the step size of loop l. Loop normalization is always legal.

In the second step, *virtual mapping*, we divide a set of array references into partitions that access independent array elements, and map each partition to a separate virtual memory. This mapping is a one-to-one mapping between the original array indices and new array indices in virtual memories, based on array access pattern information. We first analyze the data access pattern (stride and offset) of each individual array reference. Since the data access patterns of different array dimensions are orthogonal, each dimension can be treated independently. Assuming arrays accessed within their bounds, if two array references access mutually exclusive array indices in at least one dimension, they access independent array elements. In this case, we put them in separate partitions. Otherwise, we put them in the same partition, and derive a single unified data layout for them based on their common data access pattern[1]. To maximize the opportunities of parallel memory accesses, we create as many partitions as possible. The compiler rewrites each array reference so that the transformed subscript expression takes into account both the data element's virtual memory assignment and position within the newly formed array in the virtual memory. Based on the data access patterns of the code, we insert array distribution/gathering code to/from multiple memories.

In the third step, *physical mapping*, the compiler binds virtual memory to physical memory, taking into account memory access conflicts based on the array access order in the program, to exploit both memory access and instruction-level parallelism.

The algorithm in Figure 3 is most effective in the affine domain, but it can handle some non-affine array subscript expressions. For simplicity of presentation, the bulk of this paper assumes all dimensions are affine, but non-affine references are briefly described in Section 4.4. In the remainder of this paper, $\boldsymbol{M_p}$ refers to the total number of physical memories available in the system, and $\boldsymbol{M_v}$ the number of virtual memories to which the arrays in a specific partition will be mapped. The next section describes step 2 in Figure 3. Step 3 is discussed in Section 5.

4. Virtual Mapping

In this section, we present how we derive a virtual mapping for each array as outlined in Step 2 of the algorithm in Figure 3.

4.1. Analyzing Data Access Patterns

For each array reference, we analyze the data access pattern of each array dimension independently. We first transform array subscript expressions into a *canonical* form by performing constant propagation, constant folding, and algebraic simplification; *i.e.*, $A[a_1L_1+\cdots+a_nL_n+b]$. We denote an affine array subscript expression in each array dimension as a *Single Index Variable* (SIV)[2] subscript expression if there is only one non-zero coefficient. If there is more than one non-zero coefficient in the array subscript expression, it is a *Multiple Index Variable* (MIV) subscript expression. If all the coefficients are zero, it is a *Zero Index Variable* (ZIV) subscript expression.

1 A possible exception to step 2(b) is read-only arrays such that their array references access common array elements. They can be replicated to multiple memory banks if deemed cost effective. If replication is used, renaming need not modify the subscript expressions.

2 In this paper, ZIV, SIV and MIV refer respectively to zero, single, and multiple index variable subscript expression for brevity.

Basically, we identify two things from the affine subscript expression in each dimension of each array reference: a stride and an offset. An offset is simply the constant term b of the canonical array subscript expression. In the case of MIV, the subscript expression may access array elements with different strides on different loops; *i.e.*, the stride on each loop L_i is a_i. We define a stride $\boldsymbol{s_i}$ for a particular array dimension *i* of a particular canonical array subscript expression as the greatest common stride among all *n* loops; *i.e.*,

$$s_i = \begin{cases} 0, & \text{if ZIV;} \\ a_i, & \text{if SIV;} \\ \mathtt{GCD}(a_1^i, \cdots, a_n^i), & \text{if MIV.} \end{cases} \tag{1}$$

Custom data layout attempts to map to memory only those array elements that are accessed by the code. Thus, some elements may be omitted, such as in strided accesses, or accesses where the subscript for one or more dimensions is held constant. For example, the stride and the offset of array reference $A[2i{+}4j{+}1]$ are 2 and 1, respectively. We map only the odd array elements starting from array index 1.

4.2. Partitioning Array References

When two array references access mutually exclusive array elements, and thus there is no data dependence between them, we can put them in separate partitions. For example, consider array references $A[4i]$, $A[4i{+}1]$, and $A[2i]$. $A[4i]$ accesses a subset of array elements accessed by $A[2i]$, but $A[4i{+}1]$ accesses independent array elements. So, we derive a unified data layout for $A[4i]$ and $A[2i]$, and a separate data layout for $A[4i{+}1]$. The following theorem proves a key property used by the partitioning algorithm.

THEOREM 1 *Two **m**-dimensional array references A and A' access independent locations, and can be placed in separate partitions if the following condition holds:*

$$\begin{cases} b_i' \neq b_i, & \text{if both } s \text{ and } s' \text{ are zero;} \\ b_i' \bmod \mathtt{GCD}(s_i, s_i') \neq b_i \bmod \mathtt{GCD}(s_i, s_i'), & \text{otherwise.} \end{cases} \tag{2}$$

where s_i and s_i' are the strides, and b_i and b_i' are the offsets associated with two array references A and A' for each array dimension i.

PROOF. When both array references are ZIV in dimension *i*, they access independent array elements if their constant indices are different; *i.e.*, $b_i' \neq b_i$.

Otherwise, $\mathtt{GCD}(\boldsymbol{s_i}, \boldsymbol{s_i'})$ represents the greatest common stride of array indices that may be accessed by both A and A' on dimension i[3]. We prove Equation 2 by contradiction. Let two array references access the same array index in a particular array dimension in two different iterations l_i and l_i' of each loop L_i; *i.e.*, $a_1l_1 + \cdots + a_nl_n + b =$

3 If either of two array references is ZIV, $\mathtt{GCD}(\boldsymbol{s_i}, \boldsymbol{0}){=}s_i$.

$a'_1 l'_1 + \cdots + a'_n l'_n + b'$.
Rearranging terms, $a'_1 l'_1 + \cdots + a'_n l'_n - a_1 l_1 - \cdots - a_n l_n = b - b'$.
Let $s = \texttt{GCD}(a_1, \cdots, a_n, a'_1, \cdots, a'_n)$.
If we divide both sides by s, $(a'_1 l'_1 + \cdots + a'_n l'_n - a_1 l_1 - \cdots - a_n l_n)/s = (b - b')/s$.
The left hand side is an integer, since all terms are integers, and s is the common factor of $a_1, \cdots, a_n, a'_1 \cdots, a'_n$. For the right hand side to be an integer, s must divide $b - b'$. Thus, $(b - b')$ mod $s = 0 \Leftrightarrow b$ mod $s = b'$ mod s. □

The partitioning algorithm uses Equation 2 for each array dimension to divide array references into partitions. We define a common stride s_i for dimension i across k array references as follows:

$$s_i = \texttt{GCD}(s_i^1, \cdots, s_i^k) \quad (3)$$

The algorithm for partitioning the array references in a loop nest is shown in Figure 4. Initially, all array references belong to a single partition, *Set*. Then, procedure `Partition` separates array references into different partitions whenever it can prove they are accessing independent array elements using Equation 2. In a specific array dimension *i*, array references within a partition are recursively partitioned according to Equation 2 until no further partitioning is possible. The recursive function `SubPartition` derives a set of $Pmax$ possible subpartitions (some are empty). If all references are mapped to the same subpartition, then that dimenion's references cannot be partitioned, so the algorithm returns the result of partitioning the next dimension. Otherwise, it attempts to further partition a subpartition according to the current dimension.

For example, consider *Set* = $\{A[2i], A[4i{+}3], A[8i{+}1], A[8i{+}5]\}$. The common stride GCD(2, 4, 8, 8) is 2. According to the condition in Equation 2, *Set* is divided into two partitions $\{A[2i]\}$ and $\{A[4i{+}3], A[8i{+}1], A[8i{+}5]\}$. The second partition is further partitioned into two subpartitions $\{A[4i{+}3]\}$ and $\{A[8i{+}1], A[8i{+}5]\}$, since GCD(4, 8, 8) = 4 and (3 mod 4) $\neq$ (1 mod 4) = (5 mod 4). Further, the second subpartition is divided into two subpartitions $\{A[8i{+}1]\}$ and $\{A[8i{+}5]\}$. Therefore, all four references in *Set* access completely independent array elements, and are mapped to separate virtual memories.

```
Procedure Partition(Set)
Set: a set of references to an array
$m$: number of dimensions of array
$Q$: a set of partitions; i.e., $\{P\}$

$Q$ = Subpartition(Set, 1, $m$);  /* set of subpartitions of Set */
return $Q$

Procedure Subpartition(P, i, m)
P: a set of k references to an array
$Q$: a set of partitions; i.e., $\{P\}$
i: array dimension to be partitioned
m: total number of dimensions of array
$A$: an array reference in $P$
    $B[a_1^1 L_1 + \cdots + a_n^1 L_n + b_1] \cdots [a_1^m L_1 + \cdots + a_n^m L_n + b_m]$
$s_i^j$: a stride on array dimension $i$ of array reference $j$, where
    $1 \le j \le k$
$Pmax$ : maximum number of subpartitions for $P$ in dimension $i$

$k = |P|$;          /* cardinality of set $P$ */
if ($i > m$  or  $k$ = 1) return $P$
$s_i$ = GCD($s_i^1, \cdots, s_i^k$);     /* common stride for $k$ references */

if ($s_i$ = 0) $Pmax$ = max($b_i^1, \cdots, b_i^k$)
else       $Pmax = s_i$
for j = 1, $Pmax$
    $P_j = \emptyset$ /* create $Pmax$ empty subpartitions */

for each $A \in P$ {
    if ($s_i = 0$) insert $A$ into $P_{b_i}$
    else        insert $A$ into $P_{(b_i \ mod \ s_i)}$
}
$Q = \emptyset$
for j = 1, $Pmax$   {
    if ($P_j \neq \emptyset$ and $P_j \neq P$)
        $Q = Q \cup$ Subpartition($P_j, i, m$)
}
if ($Q = \emptyset$) $Q$ = Subpartition($P, i+1, m$)
if ($Q = \emptyset$) $Q = P$    /* unable to derive subpartitions */
return $Q$    /* a set of subpartitions */
```

Figure 4. Array partitioning algorithm.

4.3. Array Renaming

Once all m array dimensions are partitioned, the next step of virtual mapping is to map each partition to a distinct virtual memory independently. We rewrite array references in a partition P to represent accesses to the same virtual memory. To rename each array reference and to form the new subscript expression within a partition, we must derive the following three components for each array dimension:

1. **`suffix`**, to be concatenated to the original array name, designating the virtual memory to which this reference is mapped; and
2. **`offset`**, to be added to the loop index variables with non-zero coefficients in the subscript expressions, designating the offset within the virtual memory; and
3. **`coeff`**, a new coefficient to represent the stride of accesses in the corresponding loop after the custom layout.

We derive these components using the common stride using the common stride in Equation 3 within a partition and the offset b of each array reference.

In this section, we first describe array renaming for one-dimensional arrays. Then, we extend the approach to multi-dimensional arrays.

4.3.1. Single Dimension A specific one-dimensional array reference with ZIV, SIV, or MIV in a partition is rewritten as follows:

ZIV: $A[b] \Rightarrow A \bullet \texttt{suffix}[\texttt{offset}]$
SIV: $A[aL + b] \Rightarrow A \bullet \texttt{suffix}[\texttt{coeff}L + \texttt{offset}]$
MIV: $A[a_1L_1 + \cdots + a_nL_n + b]$
$\Rightarrow A \bullet \texttt{suffix}[\texttt{coeff}_1L_1 + \cdots + \texttt{coeff}_nL_n + \texttt{offset}]$

The new array name $A\bullet\texttt{suffix}$ uniquely identifies a partition and its corresponding virtual memory. The new subscript expression represents locations within the virtual memory.

We derive `suffix, offset`, and `coeff` for a specific array reference as follows:

$$\texttt{suffix} = \begin{cases} b, & \text{if } s = 0; \\ b \bmod s, & \text{otherwise.} \end{cases} \quad (4)$$

$$\texttt{offset} = \begin{cases} b, & \text{if } s = 0; \\ \lfloor b/s \rfloor, & \text{otherwise.} \end{cases} \quad (5)$$

$$\texttt{coeff}_i = \begin{cases} a_i, & \text{if } s = 0; \\ a_i/s, & \text{otherwise.} \end{cases} \quad (6)$$

In Equation 4, (b mod s) ensures that array references that access the common array indices are mapped to the same virtual memory.[4] In Equation 5, b is divided by stride s because s is the unit of mapping to a virtual memory. A coefficient of each loop variable is divided by s in Equation 6 to represent the stride on each loop after virtual mapping.

Theorems 2 and 3 below prove that Equations 4, 5, and 6 ensure a one-to-one mapping between the original array indices accessed by two different array references $A[a_1L_1 + \cdots + a_nL_n + b]$ and $A[a'_1L_1 + \cdots + a'_nL_n + b']$ and the array indices in the virtual memories after virtual mapping. In the case of a s=0 where the partition consists of ZIVs only, the proof of one-to-one mapping is trivial. We also omit the proof for SIV, since SIV is a special case of MIV. For this discussion, we assume array A has no aliases.[5]

THEOREM 2 *If two array references access the same array index; i.e.,* $a_1l_1 + \cdots + a_nl_n + b = a'_1l'_1 + \cdots + a'_nl'_n + b'$ *in two iterations* l_i *and* l'_i *of each loop* L_i*, then the corresponding array element is accessed at the same array index in the same virtual memory after virtual mapping; i.e.,*

1. b mod $s = b'$ mod s, and
2. $\texttt{coeff}_1 l_1 \cdots + \texttt{coeff}_n l_n \cdots + \lfloor b/s \rfloor =$
   $\texttt{coeff}'_1 l'_1 \cdots + \texttt{coeff}'_n l'_n \cdots + \lfloor b'/s \rfloor$.

`PROOF`. The proof of b mod $s = b'$ mod s follows from the proof of Theorem 1.

Rearranging terms, we know that $a'_1l'_1 + \cdots + a'_nl'_n - a_1l_1 - \cdots - a_nl_n = b - b'$.
If we divide both sides by s, we obtain $\{a'_1l'_1 + \cdots + a'_nl'_n - a_1l_1 - \cdots - a_nl_n\}/s = (b - b')/s$, which is equivalent to
$\texttt{coeff}'_1 l'_1 + \cdots + \texttt{coeff}'_n l'_n - \texttt{coeff}_1 l_1 - \cdots - \texttt{coeff}_n l_n = (b - b')/s$.
Since (b mod s) = (b' mod s), $\lfloor b/s \rfloor - \lfloor b'/s \rfloor = b/s - b'/s$.
Thus, $\texttt{coeff}_1 l_1 \cdots + \texttt{coeff}_n l_n \cdots + \lfloor b/s \rfloor = \texttt{coeff}'_1 l'_1 \cdots + \texttt{coeff}'_n l'_n \cdots + \lfloor b'/s \rfloor$. □

THEOREM 3 *If two array references* A *and* A' *access two different array indices; i.e.,* $a_1l_1 + \cdots + a_nl_n + b \neq a'_1l'_1 + \cdots + a'_nl'_n + b'$*, then the corresponding array elements are accessed either in separate virtual memories or at different array indices within the same virtual memory after virtual mapping; i.e.,*

1. b mod $s \neq b'$ mod s, or else
2. $\texttt{coeff}_1 l_1 \cdots + \texttt{coeff}_n l_n \cdots + \lfloor b/s \rfloor \neq$
   $\texttt{coeff}'_1 l'_1 \cdots + \texttt{coeff}'_n l'_n \cdots + \lfloor b'/s \rfloor$.

`PROOF`. The first case occurs when two array references are accessing mutually exclusive array indices, and thus belong to separate partitions. They access array indices with the common stride s, starting from b and b', respectively. If s does not divide the difference between b and b', array index b' is never accessed by A, and array index b is never accessed by A' in the entire iterations of loop L_i. Thus, $(b' - b)$ mod $s \neq 0 \Leftrightarrow b$ mod $s \neq b'$ mod s.

The second case occurs when an array index accessed by an array reference is also accessed by another reference in some iterations other than l'_i, since they are mapped to the same virtual memory. For example, consider $A[2i]$ and $A[2i + 4]$. $A[2]$ is accessed by $A[2i]$ when i=1, and $A[8]$ is accessed by $A[2i + 4]$ when i=2. These array elements are mapped to different locations in the same virtual memory. The proof is omitted, but is similar to the proof for the second equation in Theorem 2, substituting inequality for equality. □

4.3.2. Multiple Dimensions To rename multi-dimensional arrays, we combine the results of renaming each dimension, as will be discussed in this section. For all applicable dimensions, Equations 4, 5, and 6 from the previous section can be applied. We rewrite

4 Note that one of b or s may be a negative number. Since C compilers do not standardize on modulo arithmetic in the presence of negative numbers, we would like to clarify and avoid confusion. The result of (b mod s) should be a positive number ranging between 0 and $abs(s)$-1. If b is negative, it is defined as $(s - 1) - ((-b - 1) \bmod s)$. If s is negative, it is defined as (b mod $-s$).

5 In the presence of aliases, we would form equivalence classes of aliased objects and consider all elements of an equivalence class as accesses to the same object. In the worst case, objects with complicated aliases would be mapped to a single memory.

m-dimensional array references in an *n*-deep loop nest as follows:

$$A[a^1_1 L_1 + \cdots + a^1_n L_n + b_1] \cdots [a^m_1 L_1 + \cdots + a^m_n L_n + b_m]$$
$$\Rightarrow A \bullet \texttt{suffix}_1 \bullet \cdots \bullet \texttt{suffix}_m$$
$$[\texttt{coeff}^1_1 L_1 + \cdots + \texttt{coeff}^1_n L_n + \texttt{offset}_1] \cdots$$
$$\cdots [\texttt{coeff}^m_1 L_1 + \cdots + \texttt{coeff}^m_n L_n + \texttt{offset}_m]$$

Again, since ZIV and SIV subscript expressions are a special case of MIV, we treat them identically. The new array name in each virtual memory is now the concatenation of the original array name and a set of suffixes, $\texttt{suffix}_1, \cdots, \texttt{suffix}_m$, for each array dimension. Each $\texttt{suffix}_i$, $\texttt{offset}_i$, and $\texttt{coeff}^i_j$ is computed by Equations 4, 5, and 6, respectively, whenever array renaming is applicable for dimension i.

The partitioning decision is based on Equation 2 in Theorem 1, and following the partitioning algorithm, each partition is mapped to a unique virtual memory id. Thus, the number of virtual memories is equivalent to the number of partitions. Therefore, for each *m*-dimensional array, the cardinality of the set of distinct suffix values determines the total number of virtual memories M_v; that is,

$$M_v = |\{\texttt{suffix}_1 \bullet \texttt{suffix}_2 \bullet \cdots \bullet \texttt{suffix}_m\}|$$

If the total M_v exceeds M_p, then the physical mapping phase decides which virtual memories are mapped to the same physical memory.

The proof that array renaming is a one-to-one mapping for multi-dimensional arrays follows from the proofs of Theorems 3, and 4, and the observation that each dimension can be treated independently.

4.3.3. Array Renaming Algorithm Once all m array dimensions are partitioned, we call `Array_Renaming` in Figure 5 for each array in the code. `Array_Renaming` maps each partition to a distinct virtual memory. Based on the common stride computed in the procedure `Subpartition` in Figure 4, we compute `stride`, `offset`, and `coeff` using the Equations 4, 5, and 6.

4.4. Examples

Table 1 shows examples of virtual mapping. Two array references in Table 1(b) access respectively even and odd elements with the same stride GCD(2,2) = 2, but two different initial indices 0 and 1. According to the condition in Equation 2, each reference accesses mutually exclusive array elements, and they are mapped to two virtual memories $A0$ and $A1$. The new coefficients are both 1, and the new offsets within each virtual memory are both zeros. As such, two references in each virtual memory becomes consecutive accesses. In Table 1(b), two array references have strides 2 and 6, respectively. The common stride is GCD(2,6)

```
Procedure Array_Renaming($Q$)
$Q$: a set of partitions of array references = $\{P\}$
$A$: an array reference in $P$
    $B[a^1_1 L_1 + \cdots + a^1_n L_n + b_1] \cdots [a^m_1 L_1 + \cdots + a^m_n L_n + b_m]$
$s_i$: the common stride on array dimension $i$ in $P$

for each $P \in Q$ {
  for each $A \in P$
    for $i$ = 1, $m$ {
      if ($s_i \neq 0$) {
        $\texttt{suffix}_i = b^j_i \bmod s_i$;
        $\texttt{offset}_i = b^j_i / s_i$;
        for $j$ = 1, $n$    /* for each loop of n-deep loop nest */
          $\texttt{coeff}^i_j = a^i_j / s_i$;
      }
      else {              /* all references in $P$ are ZIVs */
        $\texttt{suffix}_i = b_i$;
        $\texttt{offset}_i = b_i$;
        for $j$ = 1, $n$    /* for each loop of n-deep loop nest */
          $\texttt{coeff}^i_j = a^i_j$;
      }
    }
    Replace $A$ with
      $B \bullet \texttt{suffix}_1 \bullet \cdots \bullet \texttt{suffix}_m$
        $[\texttt{coeff}^1_1 L_1 + \cdots + \texttt{coeff}^1_n L_n + \texttt{offset}_1] \cdots$
        $\cdots [\texttt{coeff}^m_1 L_1 + \cdots + \texttt{coeff}^m_n L_n + \texttt{offset}_m]$.
}
```

Figure 5. Array renaming algorithm.

= 2, and the two array references access mutually exclusive array elements.

The rest of the examples are two-dimensional arrays. We will consider one dimension at a time. In looking at the first dimension in Table 1(c), [2*i*] and [4*i*] are accessing common indices, but [2*i*+1] is accessing independent indices. So, we group three array references into two partitions $\{A[2i][2j{+}1], A[4i][[4j]\}$ and $\{A[2i{+}1][2j]\}$. Next, looking at the second dimension of the first partition, [2*j*+1] and [4*j*] are accessing independent indices. Thus, we further divide the first partition into subpartitions $\{A[2i][2j{+}1]\}$ and $\{A[4i][[4j]\}$. Each reference is independently renamed. Table 1(d) includes ZIV subscript expressions. In looking at the first dimension, the common stride is GCD(2,0,2,0) = 2. Thus, we divide array references into two partitions $\{A[2i][2j], A[4][2j]\}$ and $\{A[2i{+}1][2j{+}1], A[5][6]\}$. In looking at the second dimension of the second partition, the common stride is GCD(2,0) = 2, and (1 mod 2) $\neq$ (6 mod 2). Thus, the second partition is further divided into two subpartitions.

Table 1(e) shows a more complex example where the loop index variables appear in different dimensions between two references. While one reference is accessing a row of array elements, the other reference accesses a column of ar-

	Original Array References	GCD (s_1^1, s_1^2)	GCD (s_2^1, s_2^2)	Partitions			After Renaming		
				P_1	P_2	P_3	P_1	P_2	P_3
(a)	$A[2i], A[2i{+}1]$	2		$A[2i]$	$A[2i{+}1]$		$A0[i]$	$A1[i]$	
(b)	$A[2i{+}4j], A[6i{+}6j{+}1]$	2		$A[2i{+}4j]$	$A[6i{+}6j{+}1]$		$A0[i{+}2j]$	$A1[i{+}j]$	
(c)	$A[2i][2j{+}1]$, $A[4i][4j]$, $A[2i{+}1][2j]$	2	2	$A[2i][2j{+}1]$	$A[2i{+}1][2j]$	$A[4i][4j]$	$A01[i][j]$	$A10[i][j]$	$A00[i][j]$
(d)	$A[2i][2j]$, $A[4][2j]$, $A[2i{+}1][2j{+}1]$, $A[5][6]$	2	2	$A[2i][2j]$, $A[4][2j]$	$A[2i{+}1][2j{+}1]$,	$A[5][6]$	$A00[i][j]$, $A00[2][j]$	$A11[i][j]$	$A56[5][6]$
(e)	$A[i][2j{+}1]$, $A[j][4i]$	1	2	$A[i][2j{+}1]$	$A[j][4i]$		$A01[i][j]$	$A00[j][i]$	
(f)	$A[2i{+}1][3B[j]]$, $A[2i][2j]$	2	1	$A[2i][2j]$	$A[2i{+}1][3B[j]]$		$A00[i][j]$	$A10[i][B[j]]$	
(g)	$A[4i][4j \times k{+}1]$, $A[2i][2j]$	2	2	$A[2i][2j]$	$A[4i][4j \times k{+}1]$		$A00[i][j]$	$A01[i][j \times k]$	

Table 1. Examples of virtual mapping in two-dimensional multiple UGSs.

ray elements. The algorithm in Figure 4 supports such references, since only the stride and offset for each dimension is used for partitioning. Looking at the first dimension in Table 1(f), two array references are accessing common indices whenever $i = j$; GCD(1,1) = 1. So, they remain in a single initial partition. Similarly, in looking at the second dimension, the common stride is GCD(2,4) = 2. Thus, the two array references are accessing independent indices in the second dimension, so they are placed into separate partitions.

Table 1(f) and (g) show examples of non-affine array subscript expressions such as $[3B[j]]$ and $[4j \times k{+}1]$. Non-affine subscripts are not supported by the algorithm as described in this paper, since the stride must be computed differently. However, if accesses can be placed in separate partitions according to other dimensions that have affine references, then the algorithm can skip the non-affine dimension during partitioning. Further, a conservative approximation of the stride could be used to extend the algorithm for partitioning dimensions with non-affine references. For example, the exact value of $B[j]$ or $j \times k$ is unknown at compile time. But, we know that $3B[j]$ should be a multiple of 3, and that $4j \times k$ should be a multiple of 4. Looking at the first (affine) dimension in Table 1(f), we can derive separate layouts for the references. In Table 1(g), the common stride of the second dimension is GCD(4,2) = 2. According to Equation 2, the two references are divided into two partitions.

4.5. Array Reorganization

For data that is upwards exposed to the beginning of the transformed loop nest, or computed within the transformed loop nest and live on exit, it may be necessary to reorganize the data from/to a naïve layout in a single memory to/from a custom layout in multiple virtual memories. In this section, we describe how to perform this reorganization based on the data access patterns; *i.e.*, stride (s_i) and offset (b_i) in each array dimension i.

Let LB_j and UB_j be the lower and upper bound of each loop L_j, respectively. For simplicity, let's assume $s_i \neq 0$. Based on s_i and b_i in each dimension i, then, the array indices accessed by array subscript expression $[a_1^i L_1 + \cdots + a_n^i L_n + b_i]$ for the naïve layout can be formulated to

$$[(b_i \ \texttt{mod} \ s_i) + v_i \times s_i],$$

where v_i in each array dimension i is an integer such that $a_1^i LB_1 + \cdots + a_n^i LB_n + b_i \leq (b_i \bmod s_i) + v_i \times s_i \leq a_1^i UB_1 + \cdots + a_n^i UB_n + b_i$
$\Leftrightarrow \{a_1^i LB_1 + \cdots + a_n^i LB_n + b_i - (b_i \bmod s_i)\}/s_i \leq v_i \leq \{a_1^i UB_1 + \cdots + a_n^i UB_n + b_i - (b_i \bmod s_i)\}/s_i$.

For a virtual memory computed using Equation 4 on each dimension, we can construct a mapping from the naïve data layout to a custom data layout as follows:

$$\begin{aligned} &A[(b_1 \ \texttt{mod} \ s_1) + v_1 \times s_1][(b_2 \ \texttt{mod} \ s_2) + v_2 \times s_2] \cdots \\ &\quad \cdots [(b_m \ \texttt{mod} \ s_m) + v_m \times s_m] \\ &\Rightarrow A \bullet \texttt{suffix}_1 \bullet \cdots \bullet \texttt{suffix}_m [v_1][v_2] \cdots [v_m]. \end{aligned} \quad (7)$$

The inverse mapping is equivalent. By considering all possible virtual memories, and combining the results for multiple dimensions, we can map all elements from/to a standard layout in a single memory.

THEOREM 4 *Array reorganization described in Equation 7 ensures that the array references after array reorganization access the correct data in a virtual memory; i.e.,*

$$\begin{aligned} & a_1^i L_1 + \cdots + a_n^i L_n + b_i = (b_i \ \mathit{mod} \ s_i) + v_i \times s_i \\ \Leftrightarrow \ & \mathtt{coeff}_1^i L_1 + \cdots + \mathtt{coeff}_n^i L_n + \mathtt{offset}_i = v_i \end{aligned}$$

PROOF. According to Equations 5 and 6, we can rewrite the right hand side of $\Leftrightarrow$ as $(a_1^i/s_i)L_1 + \cdots + (a_n^i/s_i)L_n + \lfloor b_i/s_i \rfloor = v_i$.
By dividing both sides of the equality on the left hand side of $\Leftrightarrow$ by s_i and rearranging terms, we get $(a_1^i/s_i)L_1 + \cdots + (a_n^i/s_i)L_n + (b_i - (b_i \bmod s_i))/s_i = v_i$.
Thus, $(b_i - (b_i \bmod s_i))/s_i = \lfloor b_i/s_i \rfloor$, which is always

true, since (b_i mod s_i) is the remainder of b_i/s_i, and the result of applying a bottom operation to division b_i/s_i is the same as the result of removing this remainder from b_i first and then dividing by s_i. $\square$

5. Physical Memory Mapping

Virtual mapping creates as many virtual memories as needed to maximize opportunities of parallel memory accesses for each array in isolation, and in an architecture-independent way. In this section, we describe how to map virtual memories to a limited number of physical memories such that the exposed parallel memory access opportunities are preserved as much as possible.

To map the transformed code to a specific target architecture, we must take the following into account: (1) the number of physical memories M_p; (2) competing demands of multiple arrays; and (3) the scheduling algorithm of memory accesses. Intuitively, we want to distribute M_v virtual memories across M_p physical memory banks as evenly as possible, since it preserves the exposed parallel memory access opportunities, and minimizes the address bits required for each physical memory.

The actual memory operations that can be scheduled concurrently are affected by the physical memory mapping. We denote ΣM_v as the total number of virtual memories across all the arrays in a loop nest. If $\Sigma M_v \leq M_p$, we distribute each virtual memory to a different physical memory. If $\Sigma M_v > M_p$, some virtual memories must be mapped to the same physical memory, thereby possibly sacrificing potential memory parallelism. Some virtual memories carry a scheduling constraint such that the operations on the right hand side of an assignment statement must be scheduled before the operations on the left hand side. We map the virtual memories that carry the scheduling constraint to the same physical memory to give other less constrained virtual memories more freedom to be mapped to separate physical memories.

6. Experiments

This section presents experimental results for the previously described *custom data layout* algorithm for a set of kernel applications written in C; namely, a digital finite impulse response filter (FIR), matrix multiply (MM), pattern matching (PAT), Jacobi 4-point stencil (JAC), and Sobel edge detection (SOBEL).

The algorithm presented in this paper has been implemented and is fully automated within the DEFACTO system, which compiles C algorithms to FPGA-based systems. As compared to a conventional architecture, FPGAs have no instruction or data cache, and the microarchitecture is configured specifically for the target application. The target architecture for this experiment assumes a single FPGA with multiple external SRAM memories (8 is the default), and an external host processor that can load the data and configuration onto the FPGA, initiate its computation, and retrieve its results.

The following describes the compilation flow of the DEFACTO compiler. Within the SUIF compiler infrastructure, DEFACTO performs a set of code transformations: loop permutation, unroll-and-jam, scalar replacement, and custom data layout, to exploit instruction-level and memory access parallelism. In collaboration with commercial synthesis tools, DEFACTO performs an iterative design space search [20], using timing and area estimates derived from synthesis to determine the optimal unroll amount for each loop nest. A custom data layout is derived on the unrolled code and then incorporated into the final synthesized version of the hardware. The final optimized task is synthesized into an FPGA design.

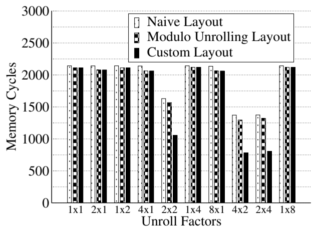
In the following experiment, we compare the performance obtained by our custom data layout with both a naïve and modulo unrolling layouts. We report results obtained from hardware simulation of the designs derived after performing these data and code transformations.

6.1. Memory Access Times

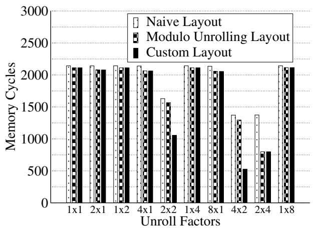
The first set of results in Figure 6, shows the time (in cycles) spent accessing memory, for each of the three layout schemes as a function of unroll factors. We show results using both four and eight memories, to see the trend as number of memories increases. With higher memory latencies, the benefits of memory parallelism increase, so we conservatively assign a low memory latency for both reads and writes of one cycle each, which is the case on our target platform when all memory accesses are fully pipelined.

In the graphs, the x-axis corresponds to different unroll factors for the inner and outer loops. For example, 1x2 refers to no unrolling on the outer loop, and an unroll factor of 2 on the inner loop.

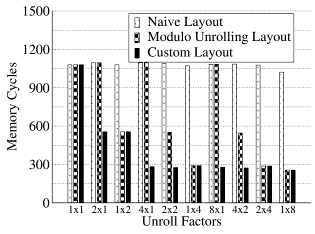
For both JAC, in Figures 6(c) and (d), and SOBEL, in Figures 6(g) and (h), which have multi-dimensional SIV subscript expressions, we see enormous decreases in memory cycles due to both modulo unrolling and custom data layout whenever the inner loop is unrolled by a factor as large or larger than the number of memories, a 75% reduction for 4 memories, and an 87.5% reduction for 8 memories. This is because unrolling the inner loop by the number of memories allows for the maximum parallel data layout to be used for arrays with the inner loop's index in their lowest dimension subscript expression. When only the outer loop is unrolled, custom data layout outperforms modulo unrolling since it distributes multiple dimensions of the accessed arrays. When the inner loop is unrolled by less than



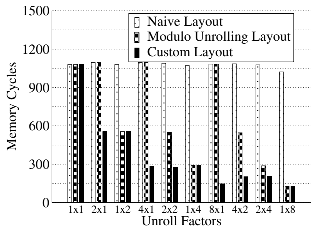
(a) FIR 4 memories



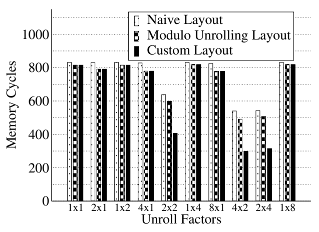
(b) FIR 8 memories



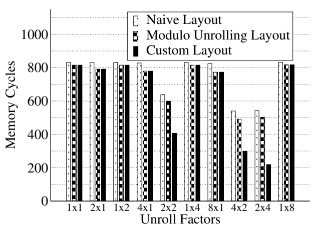
(c) JAC 4 memories



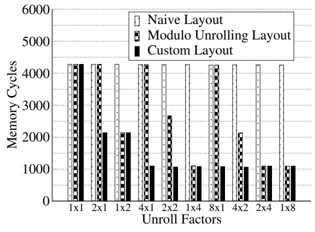
(d) JAC 8 memories



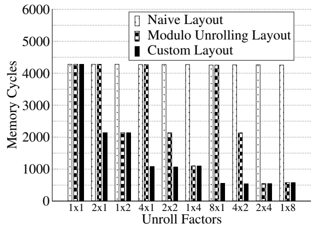
(e) PAT 4 memories



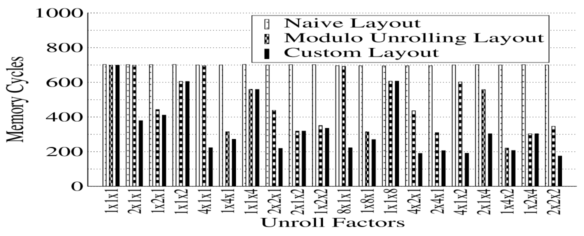
(f) PAT 8 memories



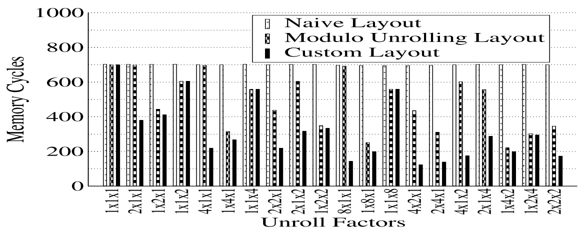
(g) SOB 4 memories



(h) SOB 8 memories



(i) MM 4 memories



(j) MM 8 memories

Figure 6. memory access times versus unroll amounts

the number of memories and the outer loop is also unrolled, although there is some performance improvement due to modulo unrolling, yielding, for example, a 50% reduction in memory access time for the 2x2 cases on 4 memories, it is not as successful as custom data layout, which still achieves a 75% reduction in this case. Results for MM are similar to those of JAC and SOBEL.

For both FIR in Figures 6(a) and (b), and PAT, in Figures 6(e) and (f), which exhibit MIV subscript expressions, both our custom layout and modulo unrolling achieve a modest improvement over naïve when only one loop is unrolled, *i.e.,* one of the unroll factors is 1.

Overall, as we go from 4 memories to 8 memories, we see the growing importance of custom data layout, since larger unroll factors for the loop or loops representing the most quickly varying dimension are needed for modulo unrolling to fully utilize the memory bandwidth of the platform.

6.2. Speedups

Now we see how the reduction in memory cycles translates into an overall reduction in execution time as compared to the naïve implementation. Rather than examining the performance relative to a large number of unroll factor choices, we consider the reduction in memory access cycles and overall speedup from the design space exploration algorithm described in [20] for 4 and 8 memories using custom data layout, as compared to a naïve layout with no unrolling and to the best modulo unrolling. We report the results in Table 2.

Even under the constraints of our target FPGA platform, we see a reduction of memory access cycles of more than 75% and fairly significant speedups for all kernels, that increases as we employ more external memories. Lower speedups were obtained for JAC and SOBEL because these kernels are highly compute bound and are not able to take advantage of the additional memory parallelism exposed by our custom layout due to the available FPGA area.

As compared to modulo unrolling, our approach yields speedups ranging from 1 to 6, for the selected unroll factors. In some cases, modulo unrolling exposes the same amount of parallelism given a sufficient unrolling of the appropriate loop, but custom layout may require less unrolling. For example, two array references $A[i, j]$ and $B[j, i]$ require a different loop to be unrolled, assuming the low-order interleaving layout. Thus, each loop must be unrolled sufficiently for each reference to exploit memory parallelism.

Program	4 mems			8 mems		
	% reduction	speedup / naïve	speedup / mod. unr.	% reduction	speedup / naïve	speedup / mod. unr.
FIR	87%	17.35	1.45	93%	22.29	1.94
JAC	77%	5.73	3.67	89%	10.25	5.93
PAT	92%	34.85	1.51	96%	46.7	2.35
SOBEL	75%	3.96	2.25	75%	3.96	3.96
MM	76%	13.37	1.86	85%	16.5	2.21

Table 2. Percentage of reduction in memory cycles and Speedups

On the other hand, custom data layout can exploit memory parallelism no matter how two loops are unrolled, thereby requiring much less unroll factors. This same is true for the MIV examples such as FIR – this is where we show a performance advantage no matter how unrolling is selected for modulo unrolling.

To be fair, these overall results show not just the benefits of increased memory parallelism, but also increased operator parallelism resulting from both loop unrolling and custom data layout. On the other hand, operator parallelism often depends on exploiting memory parallelism, so that serialization of memory accesses does not inhibit operator parallelism. Further, note that the design space exploration algorithm limits unroll factors for a variety of reasons unrelated to memory parallelism, including the limited logic capacity of the FPGA device, and balance of memory accesses and computations. Thus, speedups related to increased memory parallelism could potentially be even better on larger devices or with higher latencies to memories.

7. Related Work

Large-scale multiprocessor systems are highly affected by computation and data partitioning across processors and memory and to that end, much work [10, 11, 2, 12, 18, 5] derives coarse-grain layouts that avoid communication among processors. Kim and Prasanna [12] propose a fine-grain data distribution scheme, perfect latin squares, to map rows, columns, diagonals, and subarrays to multiple memory modules, without considering the data access pattern of the code. Thus, they support parallel array accesses only if the code is accessing consecutive locations within the square subarray. Petersen *et al.* [17] develop a type construct, and the Napa-C compiler effort [7] [8] defines C language extensions to capture the data layout. Others [16, 14, 15, 6] have developed scheduling algorithms either in software or hardware to deconflict memory hierarchy traffic.

Custom memory architectures [9, 22, 19], derived from the application characteristics, also form part of the solution to the memory-computation unit performance gap. For tiled architectures such as Raw, the Maps [4] compiler performs modulo unrolling as described earlier in the paper. Other features of the compiler include equivalence class unification for pointer analysis and static versus dynamic promotion of memory accesses.

In this work, we solve for a fine-grain data layout to decrease memory access latencies in hardware. Our user input is unannotated and we assume a fixed memory architecture. We use the application specific data access patterns to directly calculate our data layout and automatically derive the variable mapping. We replace array accesses from memory with scalar register accesses in our design for increased on-chip storage of array elements that will be reused. We leave the details of the scheduling to the Monet scheduler in our system.

8. Conclusion and Future Work

In this paper, we described an algorithm for deriving custom data layouts in multiple memory banks for array-based computations, to facilitate high-bandwidth parallel memory accesses in modern architectures where multiple memory banks can simultaneously feed one or more functional units. By examining data dependences and array subscript expressions, our algorithm automatically derives application-specific layouts in multiple memories.

As compared to solutions that reorganize computation to optimize for memory parallelism assuming a fixed data layout, our approach yields high memory parallelism for a fixed computation order by reorganizing the data. We observe greater than a 75% reduction in the number of memory access cycles and speedups ranging from 3.96 to 46.7 for 8 memories, as compared to using a single memory with no unrolling. This difference is particularly important when used in conjunction with code reordering transformations, such as loop nest transformations commonly performed on array-based computations. In addition, our data layout supports more varied data layouts than HPF-like notations (block, cyclic, block-cyclic) can do. Our compiler has more degrees of freedom in transforming code, and can thus preserve memory parallelism while accomplishing other optimization goals.

A key consideration when applying this custom data layout algorithm is the feasibility of reorganizing data in memory. Here we considered loop nest computations, but when expanding to full applications, either the compiler must use the same layout throughout the program, or it must reorganize between computations with different layouts. Depending on the architecture and the application, such a reorganization could be more costly than the performance gain from increased memory parallelism. On an FPGA platform such as ours, data reorganization in on-chip buffers has very low cost since it can be done in parallel; also, the cost of reading from and writing to external memories to perform reorganization can be reduced using parallel memory accesses for one organization order and reorganizing in parallel on chip.

A major focus of our current work is to formulate this custom data layout optimization as an interprocedural and global analysis problem, and compare the results with solutions that use efficient data reorganization. These results are currently being integrated with communication and pipelining analysis to eliminate from consideration data that need not go through memory.

References

[1] The Tera-op Reliable Intelligently adaptive Processing System (TRIPS). www.cs.utexas.edu/users/cart/TRIPS.

[2] J. Anderson. *Automatic computation and data decomposition for multiprocessors*. PhD thesis, Stanford University, March 1997.

[3] J. Babb, M. Rinard, C. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 70–81, 1999.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a compiler-managed memory system for Raw machines. In *Proc. 26th Intl. Symp. Comp. Arch.*, pages 4–15, 1999.

[5] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proc. ACM Conf. Programming Languages Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 205–217, 1995.

[6] J.-F. Collard and D. Lavery. Optimizations to prevent cache penalties for the Intel Itanium 2 processor. In *Proc. IEEE/ACM Intl. Symp. on code generation and optimization*, 2002.

[7] M. Gokhale and J. Stone. Napa C: compiling for a hybrid RISC/FPGA architecture. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 126–135, 1998.

[8] M. Gokhale and J. Stone. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 63–69, 1999.

[9] P. Grun, N. Dutt, and A. Nicolau. Access pattern based local memory customization for low power embedded systems. In *Proc. Design, Automation and Test in Europe*, pages 778–784, 2001.

[10] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions Parallel and Distributed Systems*, (2):179–193, 1992.

[11] K. Kennedy and U. Kremer. Automatic data layout for high performance Fortran. In *Proc. Ninth Intl. Conf. Supercomputing*, pages 1–24, 1995.

[12] K. Kim and V. K. Prasanna. Latin square for parallel array access. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):361–370, April 1993.

[13] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *Proc. 27th Intl. Symp. Comp. Arch.*, pages 161–171, 2000.

[14] B. Mathew, S. McKee, J. Carter, and A. Davis. Design of a parallel vector access unit for SDRAM memory systems. In *Proc. 6th Intl. Symp. High-Perf. Comp. Arch.*, pages 39–48, 1999.

[15] S. McKee and W. Wulf. Access ordering and memory-conscious cache utilization. In *First IEEE Symp. High-Perf. Comp. Arch.*, pages 253–262, 1995.

[16] P. Murthy and S. Bhattacharyya. A buffer merging technique for reducing memory requirements of synchronous dataflow specifications. In *Proc. 12th Intl. Symp. System Synthesis*, pages 78–84, 1999.

[17] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *Proc. Principles Programming Languages*, ACM SIGPLAN Notices, 2002.

[18] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proc. Principles Programming Languages*, ACM SIGPLAN Notices, pages 101–112, 2002.

[19] P. Slock, S. Wuytack, F. Catthoor, and G. de Jong. Fast and extensive system-level memory exploration for ATM applications. In *Proc. 10th Intl. Symp. System Synthesis*, pages 74–81, 1997.

[20] B. So, M. Hall, and P. Diniz. A compiler approach to fast design space exploration in FPGA-based systems. In *Proc. ACM Conf. Programming Languages Design and Implementation*, pages 165–176, June 2002.

[21] B. So, H. Ziegler, and M. Hall. A compiler approach for custom data layout. In *Proceedings of the 15th annual workshop on Languages and Compilers for Parallel Computing*, 2002.

[22] M. Weinhardt and W. Luk. Memory access optimization and RAM inference for pipeline vectorization. In *Proc. 9th Intl. Workshop, Field-Progammable Logic and Applications*, pages 61–70, 1999.

[23] M. Wilkes. The memory wall and the cmos end-point. In *Comp. Arch. News*, number 4 in ACM SIGARCH, 1995.

[24] X. Zhuang, S. Pande, and J. Greenland. A framework for parallelizing load/stores on embedded processors. 2002.