Trading off Latency for Memory Bandwidth in Spatial Architectures using the Polyhedral Model

Benoît Meister
Reservoir Labs, Inc.
meister@reservoir.com

***Abstract*—We present a fast algorithm for transforming loop nests represented in the polyhedral model, which have been placed onto a spatial grid of processors. The goal of the transformation is to avoid broadcasts, which can put extra pressure on remote memory bandwidth, and turn them mainly into neighbor-to-neighbor reuse. The transformation preserves existing parallelism and locality properties of the original program as much as possible, and in particular does not reduce its amount of parallelism.**

***Index Terms*—Spatial architectures, polyhedral model, performance, memory bottleneck, data transfer optimizations**

I. Background

Some computer architectures – often called *spatial* architectures [4], [11], [10], [20], [21] – are based on a grid of processing elements (PEs) which have the ability to transfer data to and from PEs that are physically close to them. These PEs are called *neighbors* to the PE under consideration. We are interested in turning broadcast communications among such PEs into neighbor-to-neighbor communications. This should obviously benefit machines that do not have accelerated broadcast mechanisms, or for which neighboring communication is faster than remote processor communication. The method we present here can also help with bandwidth-limited platforms. PEs can typically transfer data from remote memories on a limited bandwidth. In highly parallel systems, in which there are many PEs in the PE grid, there is a significant risk of saturating the bandwidth to remote memories.

Saturating the bandwidth results in a reduction of both program performance and performance predictability. When data is being reused across PEs, there is an opportunity to reduce the amount of data transfers from remote memories by turning several simultaneous loads of the reused data from remote memory into a small number of loads (typically one) from remote memory, followed by a series of neighbor-to-neighbor transfers. Here we present a technique to automatically reschedule operations such that the neighbor-to-neighbor transfers can happen. We model the multiple loads from remote memory to PEs as a broadcast, i.e., a (roughly) simultaneous use of the reuse data across a set of PEs. Our method turns such simultaneous use of reused data into consecutive uses of the data by neighbors, the neighbors of these neighbors, etc. When data is used consecutively among neighbors, the data can be transferred (between the consecutive uses) directly between neighbors, as opposed to being brought in repeatedly from remote memory. Our method achieves this by:

- identifying broadcasts, for each data set accessed by the loop program, as a linear subspace of the PE grid called broadcast space;
- modifying the time at which PEs access the reused data, in such a way that PEs in the broadcast space access the reused data in sequence, where they were originally accessed in parallel; and
- organizing the sequence such that neighboring PEs access the reused data consecutively.

II. Principle

A. Broadcast space

Operations in the polyhedral model are represented by statements, to each of which is associated an iteration domain. Iterations are mapped to processors through a placement function.

Let f be a $\mathbb{Q}^n \rightarrow \mathbb{Q}^m$ linear function. Let us consider a reference $X[f(I) + f_0(N)]$ to m-dimensional array X happening in a nested loop, where $I \in \mathbb{Q}^n$ represents the set of loop counters and $N \in \mathbb{Q}^p$ is an affine (i.e., linear plus a constant) function of loop-invariant expressions (typically referred to as "parameters").

$X[f(I) + f_0(N)]$ results in a *broadcast* if and only if a given element of X is used by a subspace of the PE grid at the same time. When looking at loop programs, the notion of time maps to loop iterations. Iterations that contribute to a broadcast use the same element of X through f (i.e., they belong to f's *reuse space*) across a subspace of the processor grid at a fixed time. The *placement space* represents iterations that are mapped to different processors in the grid at a given time.

Let $Pl(I)$ be the placement function associated with a polyhedral operation. A (possibly multidimensional) placement function defines a mapping between any given loop iteration of a statement and the coordinates of the PE in the PE grid that will execute it. The placement function usually has one dimension (i.e., one function) per dimension of the PE grid.

Mathematically, the spaces above are defined as follows.

Let D be the iteration domain associated with an array reference f. f's reuse space $R(f)$ is defined as the iterations that use the same element of X in D:

$$R(f) = Ker(f) \cap D$$

Where $Ker(f)$ is the kernel of f, i.e., the iterations space $\{I \in \mathbb{Q}^n : f(I) = 0\}$. The analysis we present here relies on

subspaces, and hence it is sufficient to retain the equalities E of D in the definition of the reuse space:

$$R(f) = Ker(f) \cap E \tag{1}$$

A polyhedral operation's *placement space* is defined as iterations that map to different processors for a given time:

$$P(Pl) = Span(Pl)$$

A more general definition of the placement space can be considered. We may want to avoid broadcasts only along a given dimension of the processor grid, in which case we'd look at the span of the corresponding rows of Pl.

The broadcast space is the subspace of iterations reusing the same element of X across processors, i.e., the intersection of $R(f)$ and $P(Pl)$.

$$B(f, Pl) = Ker(f) \cap Span(Pl) \cap D$$

B. Transformation

The goal of broadcast elimination is to reduce the dimensionality of the broadcast spaces (one space per array reference). We will see throughout this paper that this can be done by combining time dimensions into the placement dimensions. This paper defines these terms and describes a family of algorithms to automatically detect and remove the memory access patterns that lead to broadcasts.

The basic idea is to find a time vector that is independent of the broadcast space, and combine a vector of the placement space with it. The method automatically selects a vector that is independent from the placement space, which will also (by definition of the broadcast space) be independent from the broadcast space, and selects appropriate vector of the placement space to combine with.

We'll use the following matrix multiply loop nest as our running example.

Example 1.
```
for i = ...
   for j = ...
      for k = ...
         C[i][j] += A[i][k] * B[k][j];
```
with placement $Pl = (i, j)$

Placement space is the span of i and j, i.e. in constraints form:

$$P = \{k = 0\}$$

Reuse spaces are:

$$\begin{aligned} R_C &= \{i = 0; j = 0\} \\ R_A &= \{i = 0; k = 0\} \\ R_B &= \{j = 0; k = 0\} \end{aligned}$$

Hence broadcast spaces are:

$$\begin{aligned} B_C &= \{0\} \\ B_A &= \{i = 0; k = 0\} \\ B_B &= \{j = 0; k = 0\} \end{aligned}$$

B_A is a line along j, and B_B a line along i, which means that A is broadcast along j and B is broadcast along i.

Now let's introduce a j skew in k (here, k is the only time dimension available), with the following change of variables:

$$\begin{array}{ccc} j' = j & \Leftrightarrow & j = j' \\ k' = k - j & & k = j + k' \end{array} \tag{2}$$

This results in the following program:

```
for i = ...
   for j = ...
      for k' = ... (skewed bounds)
         C[i][j] += A[i][k' + j] * B[k' + j][j];
```

with Placement $Pl = (i, j)$.

The placement space hasn't changed, but the reuse spaces have:

$$\begin{aligned} R_C &= \{i = 0; j = 0\} \\ R_A &= \{i = 0; k' + j = 0\} \\ R_B &= \{k' + j = 0; j = 0\} \end{aligned}$$

As a result, broadcast spaces have changed as well:

$$\begin{aligned} B_C &= \{0\} \\ B_A &= \{0\} \\ B_B &= \{j = 0; k' = 0\} \end{aligned}$$

None of A's elements is broadcast anymore in the resulting program. Another skewing involving i would similarly eliminate the broadcast of elements of B.

III. Methods

Now that we understand how to detect broadcasts and have an idea of how to remove them, we will first look at how to compute broadcast-eliminating transformations for one array reference. Then, we will define heuristics for reducing broadcast in the case of multiple references, and even the general case of imperfectly nested loops with arbitrary number of statements. Our technique supports PE grids that are connected to their neighbors in arbitrary subsets of the canonical directions.

A. General method

The broadcast space, which is the intersection between placement space and reuse space, has more than zero dimensions whenever the number of independent equalities from the placement space and the reuse space is less than the dimensions of the iteration space. There is a broadcast whenever the dimension of the broadcast space is non-zero.

The dimension of the broadcast space is given by $n - rank(B)$, where B is made by concatenating the normals to the hyperplanes defined by the equalities of the reuse and placement spaces. When the broadcast space has more than zero dimensions, we can reduce it when the normals to the placement space are not independent from the normals to the reuse space. Broadcast elimination reduces the dimensionality of the broadcast space by modifying the iteration space (through re-scheduling), such that either the placement space or the reuse space (or both) is modified. The broadcast space

is reduced when the intersection of the resulting reuse and placement spaces is defined by more independent normals.

Basically, we want to turn some dependent normals into independent ones. By definition, the placement space spans the iterations used to represent space in a space-time mapping of iterations. Hence, the normals to the placement space depend upon time dimensions.

Following this idea, we reduce the dimension of the broadcast space by the following procedure:

1) compute the space V of vectors independent from B (i.e., $Ker(B)$),
2) go through the normals of the reuse space R.
   a) Each time a normal r is detected as dependent from the normals to P and the normals of R that were already visited in step 2, transform r into $r' = r + v$, where v is a non-zero vector in V.
   b) Substitute r with r' in R

The set of substitutions $r' = r + v$ directly defines a transformation from the initial iteration space to a transformed iteration space, which reduces the broadcast space by as many dimensions as there were substitutions.

In the running matrix multiplication example, we would remove the broadcast induced by accesses to A by forming B_A as the concatenation of the normals to the reuse space (represented as rows):

$$NR_A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

with the normal to the placement space:

$$NPl = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$$

giving (after removing redundant vectors):

$$B_A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Step 1 of the above algorithm computes the kernel V_A of B_A, which is $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$. Step 2 goes as follows:

- Select the first vector from NR_A: $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$ as r. Since it is independent from the normal to the placement space NPl, there is nothing to do.
- Select the second vector from NR_A: $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ as r. It is dependent on the normal to the placement space NPl. Hence we compute $r' = r + v$, where v is in the space defined by V_A. Here we can use either $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$ or $\begin{pmatrix} 0 & -1 & 0 \end{pmatrix}$. To match our running example, let us choose $\begin{pmatrix} 0 & -1 & 0 \end{pmatrix}$.

Hence, r' is defined as:

$$\begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix} =$$

$$\begin{pmatrix} 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} =$$

$$\begin{pmatrix} 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

Or in short:

$$k' = k - j$$

B. Desirable properties for a broadcast-eliminating transformation

An absolute necessary property of the transformation is that it should be valid, i.e., that it preserves dependences.

However, we are considering broadcast elimination as a polyhedral optimization within a mapping process, i.e., a sequence of loop optimizations. Hence, a desirable broadcast-eliminating transformation should try to preserve existing schedule properties as much as possible, in order to minimize the undoing of previous optimizations. In particular, scheduling of statements with respect to each other should be preserved, in order to preserve locality. More specifically, the fusion-fission structure of the schedule should remain intact, and the same schedule transformation should be applied to any common loops among statements.

The transformation should preferably not introduce strides/steps, especially in the placement space (we want to create neighbor-to-neighbor reuse without hops). One way to reduce artificial strides is to restrict transformations to be unimodular. This maintains all iteration points on the original lattice of integer points. However, unimodularity is not sufficient to obtain neighbor-to-neighbor communication patterns, as we will see below.

Finally, a re-scheduling algorithm whose computational complexity scales well with the number of statements and loop dimensions is preferred.

C. Validity

Broadcast elimination is a scheduling transformation, which means that it defines a transformation from the original iteration space of each statement to a new one. A general validity criterion is that the lexicographic sign of dependences is not modified by the transformation. For each pair (s, t) of statements linked by a dependence D_{st}, in which $D_{st} \Rightarrow I_s \preceq I_t$, let $I'_s = \theta_s I_s$ (the schedule applied to the source statement s) and $I'_t = \theta_t I_t$ (the schedule applied to the target statement t). We need to preserve:

$$I'_s \preceq I'_t \tag{3}$$

Hence, a general search for a valid solution can be formulated as a series of Integer Linear Programs (ILPs), for each r'. However, ILPs do not present some of the important desirable properties of Section III-B: unimodularity is a non-linear constraint, and ILPs don't scale well with the number of statements and loop dimensions. Note that preserving fusion-fission structure is trivial, as it means maintaining the "beta dimensions" of the schedule. Beta dimensions [8] are used to enforce fission between the iteration domains of polyhedral statements, irrespective of the affine loop schedules found for these statements.

In the next section, we define a strategy that reduces the search space of scheduling transformations, in a way that reduces broadcasts while meeting the desired properties of Section III-B.

D. Unimodularity

A relatively easy way to always build correct unimodular schedules is to make them a lower-triangular with 1 diagonal elements (which we'll refer to as LTOD schedules). Having ± 1 diagonal elements makes it unimodular, while having positive diagonal elements preserves lexicographic sign, and hence ensures correctness (provided, as we mentioned earlier, that the beta coordinates are also unchanged). Hence, when forming schedules that have this form, no extra dependence checking is necessary. However, this is not powerful enough as it prevents us from reducing some broadcasts, as illustrated in the following example.

Example 2.
```
for k = ...
   for i = ...
      for j = ...
         C[i][j] += A[i][k] * B[k][j];
```
with placement $Pl = (i, j)$.

Broadcast is the same as in Example 1, which can be eliminated using the following schedule, which is not lower triangular.

$$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

E. Minimal per-statement distortion

According to [19], unimodular matrices can be defined as arbitrary combinations of elementary transformations:

1) permuting rows,
2) multiplying rows by -1, and
3) subtracting an integral multiple of a row from another.

Since we aim to stay as close as possible to original schedules, multiplying by -1 is unnecessary, as it would effect a loop inversion. The same is true for permuting rows, which would cause loop permutation. Hence, we are looking for a composition of legal subtractions of a row (say the i^{th}) from another (say j^{th}), which can each be written as:

$$\mathbb{1} - \alpha\delta_{ij} \tag{4}$$

where $\mathbb{1}$ is the identity matrix, α an integer, and

$$\delta_{ij}[x, y] = \begin{cases} 1 & \text{if } \{x = i, y = j\} \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Since we want to minimize distortion from the existing schedule, and avoid the introduction of hops in the communication patterns, we force $\alpha = 1$. As a result, for any given statement, a broadcast-eliminating transformation is fully defined by a set of distinct pairs i and j. From now on, we will call i the "target" dimension.

F. Minimal inter-statement distortion

Another way the original schedule properties can be significantly distorted is by defining conflicting schedules coefficients for pairs of statements that originally share the dimensions corresponding to these coefficients. Doing so can modify data locality significantly, and may affect parallelism properties including permutability.

At this stage, we could consider that eliminating broadcasts is more important than preserving data locality, or the opposite. A variety of heuristics, for instance based on cost functions and including brute force search, can be applied to find an optimal trade-off between broadcast elimination and the preservation of other performance-related properties.

Here we explore some of the most practical ones, with the assumption that we want to minimize the impact of the broadcast elimination transformation on the properties of the existing schedule. The main choice here is to produce consistent schedules. This means that for each pair (s, t) of statements that share a common beta (i.e., fusion-fission) coordinate at a given dimension k, the k^{th} row of the schedule matrices θ_s and θ_t are the same. The fusion-fission structure of statements is conveniently represented in R-Stream by their beta-tree [16], in which one node is defined per schedule dimension. Two statements share a schedule dimension at dimension k if and only if their ancestor at level k in the beta-tree is the same node. Because fusing loop dimensions at level $l > k$ is only possible when they are fused at level k, it can easily be seen that the structure of the beta nodes is a tree (hence the name beta-tree).

Note that finer trade-offs can be made between the need to maintain existing schedules and the need to perform broadcast elimination. In fact, maintaining consistent transformations can force the transformation of statements that do not have any broadcasts. Since we are trying to preserve existing schedule properties, one sensible way to deal with that case is to decide whether to perform the (consistent) broadcast elimination or not, rather than producing inconsistent transformations.

For simplicity, we will assume that placement is consistent across statements, i.e., that the portion of placement functions involving the k common dimensions are the same for s and t. The algorithms we propose can be generalized to placement that is inconsistent across functions by grouping statements by consistent placement functions.

Finally, the algorithms we are presenting work equally well on tiled or non-tiled code. For tiled code, the optimization pass we implemented is able to eliminate broadcast among inter-tile loop dimensions or intra-tile. An assumption here is that each dimension of placement functions is completely expressed in either the inter- or intra-tile dimensions, but not a mix of both. The decision to perform one or the other is based on properties of the machine model, which defines which processing level(s) of the architecture are spatial, i.e., have neighbor-to-neighbor communication capabilities.

G. Legal unimodular transformations

We have seen in Section III-D that LTOD schedules are always legal. Hence, we can check for the existence of an LTOD schedule. If it exists, it is legal. Similarly, we can look for a schedule (as in Section III-I), and test if it is LTOD.

If no LTOD solution is found, we can take advantage of the fact that we are starting from an already placed program to devise a scalable rescheduling algorithm.

At a high level, parallelization is the process of mapping independent computations to different computing elements. In terms of dependences in a placed program, this means that for some D_{st} dependences, we have $Pl(I_t) - Pl(I_s) = 0$. By linearity of Pl, any linear combination P of the row vectors of the Pl matrix (which represent dimensions of the placement function) satisfies

$$P_t I_t - P_s I_s = 0 \tag{6}$$

As a result, we can add or subtract any P to an existing schedule without invalidating these dependences, since we have:

$$(P + \theta_t) I_t - (P + \theta_s) I_s = 0, (I_s, I_t) \in D_{st} \tag{7}$$

In our running matrix multiplication example, all dependences respect (6). i, j and any linear combination of them can be subtracted from another iteration dimension to eliminate broadcasts in such a way that the transformation is unimodular, validates dependences and introduces minimal strides.

In fact, the two transformations we use are

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

which compose as:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}$$

However, not all dependences respect (6). Outer schedule dimensions often validate some dependences, allowing inner schedule dimensions to expose parallelism among the remaining non-validated dependences.

Another class of dependences can be ignored: the one that are validated by the fusion-fission structure of the program (beta coordinates), which we are not modifying. Basically, at a given schedule dimension k, we can ignore dependences between statements that do not share the same node of the beta-tree at dimension k.

Now let us consider the remaining kind of dependences, which constrain the set of valid combinations of Pl. At a given schedule dimension k, we basically need to find a schedule for all statements that have the same beta node at dimension k that validates the existing pairwise dependences. We are looking for consistent schedules, i.e., the schedule to find is the same for all these statements. The solution space is hence much more restricted than that of a traditional scheduling problem [7], [2], in which an independent schedule is found for each statement. Because the schedule is the same for each statement under consideration, we can decompose the problem by dependence. Hence, we can find a *per-dependence* set of valid schedule using Farkas Lemma, in the way of [7], and then find our schedule in the intersection of these valid schedule sets.

One additional aspect, which reduces the size of each per-dependence problem, is that we are looking for the same schedule for the source and target statements.

An extra-scalable but less powerful algorithm can be formulated, which abandons the search for P whenever it finds a dependence that isn't already validated by the fusion-fission structure and that doesn't respect (6).

A remaining question is which loop dimensions the rows of Pl should be subtracted from, i.e., the target dimensions.

H. Time dimensions

Let us define time dimensions as iteration space dimensions whose canonical vector is independent from the placement space. These are particularly interesting since they represent a multi-dimensional notion of time. Per the algorithm in Section III-A the schedules we are looking for will by definition be independent from the placement space. Hence, the schedules we are looking for will necessarily include time dimensions. Also, reuse occurs when a particular data set is accessed at different times, hence reuse spaces always involve time dimensions. We basically want to modify time dimensions that are involved (as normals) in the reuse space, because they are the only ones through which we can modify the reuse space.

The significance of various time dimensions are not the same. Reuse along a time dimension means that the dataset touched by the loop dimensions inner to it are reused [14]. Hence, it is important to consider locality of said dataset with respect to the targeted PE memory. Basically, if the reused data set needs to be evicted and reloaded into the targeted memory from one iteration of the considered time loop to the next, there is no point in eliminating broadcasts using said time loop, because it would still result in having every PE in the original broadcast transfer the data set with the remote memory.

Hence, only the innermost relevant time dimension should be used to turn broadcasts into neighbor-to-neighbor communications. Tiling can be performed in a way that the dataset accessed by intra-tile dimensions dimensions is reused (or mostly reused) within the tile. When this happens, the target dimension is uniquely defined as the innermost inter-tile time dimension. Otherwise, the optimal target dimension should one that defines a dataset that is significantly reused from one iteration of the target dimension to the next.

By choosing time dimensions on a statement basis, it is possible to define more than one time dimension within a group of statements which share (in the beta-tree) time dimensions. Each time dimension defines a broadcast-eliminating transformation, which needs to be applied to all the statements that share the time dimension in the beta-tree.

I. Algorithm

Since we want to produce consistent schedules, the algorithm can directly work on the beta-tree. In summary, for statements that require broadcast elimination, we need to:

1) Select a time dimension. Such a dimension corresponds to a node in the beta-tree, which we will call the statement's *time node*.

2) For each time dimension among the parent nodes of each statement in the beta-tree:
    a) Define a legal linear combination of the rows of the placement function that is a combination of unimodular transformations as defined in Equation 4.
    b) Apply the transformation consistently, i.e., to all the statements that share the time dimension in the beta-tree

This process can be repeated until all the broadcasts that need to be eliminated and can be eliminated are eliminated.

J. Partially spatial architectures

We have implicitly assumed that all dimensions of the targeted architecture are spatial, i.e., a PE can communicate with its neighbors in all directions. However, this algorithm can be straightforwardly extended to partially spatial architectures, in which such neighbor-to-neighbor communication is not available in all directions. The algorithm is simply extended by only considering the placement functions that correspond to the spatial dimensions. It is also possible to restrict the removal of broadcasts to the ones that occur along the canonical directions of the PE grid, by performing consecutive broadcast eliminations based on each dimension of the placement function at a time.

IV. Evaluation

The evaluation method we propose here focuses on the ability to turn remote-to-local memory transfers to neighboring ones. It is meant to be independent from which type of memory (scratchpad, different cache parameters) is used, and from the type of inter-processor data transfer optimization used (examples of these can be found in [1], [3]).

We created a machine model representing a 2-d grid of 16×16 cores with a 1KB local memory, and used R-Stream to map a 1024×1024 matrix multiplication code to it. Two versions of the machine model were used, one that declared the core grid as "spatial" (capable of neighbor-to-neighbor transfers) along both dimensions, and one that didn't. R-Stream's broadcast elimination acts on spatial architectures only.

We hand-instrumented the codes to turn any read and write to an array element into counting code. Instead of values, array elements maintain a state that tells which core accessed it last and when. The "when" is counted in terms of steps. The code is tiled to fit the core's local memory. Time is incremented (a "step") each time the cores have executed one tile. Assuming a mesh network, counting was performed as follows: (a) when data was never accessed, or when it was accessed more than one step ago, it's a remote access; (b) when data was accessed by the same PE in the current or previous step, it's a local access; (c) when data was accessed in the previous step by one of the 4 cardinal neighbors, it's a neighbor access. The results are represented in Figure 1. A per-core count of memory accesses (not showed here for lack of space) would show that the remote accessed are evenly distributed when broadcasts are not eliminated (around 15M), and overwhelmingly concentrated on two edges of the grid (around 5M on one edge, 10M on the other, vs. 4K on the other cores) when they are. The number of local accesses doesn't change significantly, and remote accesses are turned into neighbor accesses by enabling broadcast elimination. Unfortunately, we haven't yet automated this code instrumentation process, which prevented us from showing focused results for more benchmarks.

Version	w/ broadcasts	w/o broadcasts
Remote accesses	12.25%	0.81%
Neighbor access	0.06 %	11.72%
Local accesses	87.39%	87.47%

Fig. 1. Memory accesses, before and after broadcast elimination

V. Related work

The problem of turning non-uniform communication patterns (including broadcasts) into neighbor-to-neighbor communications was addressed for the case of systolic arrays, in which transfers were operated at a granularity of a processor cycle. Every time, some delay is introduced so as to pass one value to its neighboring computations.

The main polyhedral technique for this is called uniformization [18], which decomposes non-uniform dependences into a sum of uniform ones. We discuss this approach, the closest to ours in our view, in more detail in section V-B

Variants of uniformization were also proposed [9] in independent attempts to remove broadcasts as well as "computational broadcast", the redundant definition of array elements across the systolic array. In the case of computational broadcasts, the code is converted to single-assignment, where only one PE defines the array element and passes it on to its neighbors.

Still in the context of systolic arrays, Leiserson et al [13] consider a system of functional units linked by registers. Such a system is represented by a graph, whose edge weights represent delays between pairwise functional units. Similarly as here, broadcasts can happen in this type of computation, and the authors present *retiming*, a way to shift edge weights to modify the timing at which data originally broadcast is now passed from functional unit to functional unit. The authors show that retiming can always be applied along a depth-first spanning tree of the graph to remove broadcasts.

While describing a placement algorithm for distributed computers equipped with efficient broadcast mechanisms, Platonoff [17] mentions a case when reuse patterns turn out to be uniform, in which case the use of broadcast is not recommended. The problem is different from the one addressed here, where we trade remote memory accesses for neighboring memory accesses. We also start from code whose placement (or distribution) is known, whereas [17] is looking for a placement function.

Loechner et al [15] proposed a characterization of broadcasts by looking at pairs of references associated with inter-statement dependences. The authors look for an optimal

schedule and placement (called *allocation* in the paper) by reducing non-constant, non-broadcast communication patterns. Here again, the target architectures they consider possess efficient broadcasting capabilities, and they are not trying to eliminate broadcasts.

A. Kung-Leiserson

A famous paper by Kung and Leiserson [12] presents a set of implementations of array computations on systolic networks. A mapping of matrix multiplication, LU decomposition, and triangular system solve are presented on a two-dimensional hexagonal network (reproduced on Fig. 2), and matrix-vector multiplication is mapped to a 1-dimensional mesh. It is worth looking at some of the main differences between the mappings obtained by Kung and Leiserson on one hand, and ours on the other hand.

First, the method presented here is a general basis for turning broadcasts into neighbor-to-neighbor communications. It is not specialized to systolic arrays, and in particular, it does not incorporate constraints on the number of ports, or amount of data that can be transferred at each task iteration (i.e., each cycle in the case of a systolic array), in any given direction. While we conjecture that our method can be extended to incorporate such constraints, such extension is out of the scope of this paper.

Another noteworthy difference is that our method prefers the production of unimodular transformations. Kung and Leiserson's examples, in contrast, seem to be based on non-unimodular placement of computations. Non-unimodularity has a direct, negative impact on the utilization of the PE grid. For instance, in Kung and Leiserson's examples, computations are placed on a lattice of determinant 2, resulting in only about half the PEs working at any given time. Our method contrasts with this in that it endeavors to produce unimodular transformations, which maximize the PE grid utilization.

Finally, all the mappings in Kung and Leiserson force all arrays to flow through the interconnection network. As a result, they seem to miss opportunities to *not* transfer data that would not be broadcast anyway. To see this, compare the matrix multiplication example developed in this document with the Kung and Leiserson implementation. In the former, elements of C are computed entirely while a set of rows of A and columns of B pass through the PE grid. In the latter, however, each pass of A, B, and C only produce a partial summation of C, which means that the result of such partial summation has to be reinjected later into the systolic array, potentially creating more communications.

Another work that relates to the technique presented here is Even's use of a representation of synchronous circuits as space-time circuits to prove that retiming preserves testability of a synchronous circuit, and to define valid initial register states after retiming is applied [5]. The main common point with our technique is that it maps elementary computations (which, in Even's case is a gate, and in our case can be anything from an instruction to an arbitrary-sized task) to space and time.

B. Quinton-Van Dongen

A great paper by Quinton and Van Dongen [18] addresses the correct and efficient parallelization of systems of linear parameterized recurrence equations (SLPRE) to systolic architectures. A technique called *null-space pipelining* is introduced as part of the *uniformization* process, which decomposes non-uniform dependences into uniform ones. To some degree, the technique presented here can be considered a polyhedral scheduling-based extension of the null-space pipelining idea to multi-core architectures.

A first major difference comes in the goal and input problem. We start from a legal schedule and produce another legal schedule, while [18] starts from a feasible SLPRE and finds a legal schedule. Also, we use placement information to focus on actual broadcasts, and minimize the amount of references that need a transformation, while [18] starts from what we call the reuse space, potentially resulting in more skewings. Our algorithm also doesn't have any restrictions on the reuse space being processed, and can schedule directly without creating uniform dependences.

Finally, linear recurrence equations rely on a dynamic single-assignment (SA) property of array elements. This differs from the representation used here, in which an array element can be defined any number of times. While the conversion between both representations is possible, it does not come for free. Besides being fairly costly in itself [6], it is usually necessary to go back to the non-single-assignment (NSA) form because arrays corresponding to SA can easily have impractically large sizes. The choice of a schedule in SA form can severely limit the ability to contract these arrays, and can again result in impractically large arrays in the resulting NSA form. As a result, null-space partitioning restricts what can be done by an independent scheduler. This contrasts with our algorithm, which preserves locality and parallelism properties obtained (upstream) by an independent scheduler as much as possible.

Since our method isn't restricted to SA, it also handles broadcasts on written references (a.k.a. compute broadcasts). It is also more widely applicable, to fine- and coarse-grained computer architectures.

VI. Topological considerations

In this section, we discuss how the presented technique applies to various interconnection network topologies. We start with known non-hierarchical ones, and then discuss the relevance of our technique to hierarchical parallel computing architectures.

A. Non-hierarchical

The optimization presented here apply to a set of processors represented in an n-dimensional space (the placement function being n-dimensional). In theory, this can be applied to a theoretical n-dimensional grid of processors with an infinite number of processors along each dimension. In practice, we always end up with a finite number

1) (n-d) Mesh: The most straightforward realization of such a grid is obtained by bounding the processor coordinates along all the gird dimensions: we get a n-dimensional mesh. Without loss of generality, and also because it's the general practice, we can assume that the lower bound on any processor grid coordinate is zero. The grid is bow bounded, and it is represented as a hyper-rectangular polyhedron in the non-negative quadrant of the processor grid space (a zero lower bound and an upper bound corresponding to the number of processors in the grid along each dimension).

A trivial but uninteresting bounding case happens when the upper bound is also zero. In this case, we cannot parallelize code across this dimension, and we instead ignore this dimension, which makes us look at a (n-1)-dimensional processor grid. Other than that case, bounding does not change anything to the considerations in our technique. Given a grid size $0 \leq G < S$, the n-dimensional placement function $Pl(I)$ becomes implicitly

$$Pl'(I) = Pl(I) \bmod S \tag{8}$$

, where the modulo (integer division remainder) operation is applied element-wise.

2) Torus: Let us take a closer look at the mesh placement function defined in Equation 8. A good way to represent this mapping to a bounded processor grid is to represent it as a bijection:

$$Pl(I) = \lfloor Pl(I)/S \rfloor + Pl(I) \bmod S \tag{9}$$

which we can materialize by two new placement sets of dimensions, Pl' and Pl":

$$Pl(I) = Pl"(I) + Pl'(I) \tag{10}$$

Since the processor grid is bounded, Pl'(I) is the only part of Pl(I) to represent actual placement (i.e., processor coordinates). Hence $Pl"(I)$ represents sequential(ized) loop iterations, i.e., they become time dimensions.

Let us assume that the code is organized in tasks, i.e., a fixed number of outermost loops (called "inter-task loops") iterate across tasks, and the remaining inner loops ("intra-task loops") iterate within a task. Let us also assume that the placement function distributes tasks across the targeted processor grid. This is a general description of a parallelized loop nest, including when the loop nest is placed on a synchronous circuit (in which case the number of intra-task loops is zero). There is reuse of data across the edge of dimension k of a torus if and only if increasing the innermost inter-task loop can result in an increase of Pl"(I) by one along dimension k. Let D the iteration domain, d the dimension of the innermost inter-task loop, and let $\mathbb{1}_d$ represent the canonical unit vector along dimension d. We are looking at the k^{th} dimension $Pl(I)_k$ of the placement function $Pl(I)$.

Given a program optimized with the technique presented in this paper, which produces neighbor-to-neighbor reuses on an unbounded processor grid, we can formulate the above condition as an existence of an integer point in the following polyhedron:

$$\exists I, l \in E_k(I,l) = \begin{cases} I \in D \\ I + \mathbb{1}_d \in D \\ Pl(I)_k = S_k l + S_k - 1 \\ Pl(I + \mathbb{1}_d) = (S+1)l \end{cases} \tag{11}$$

A simpler but less inclusive test tells us if an increment of the innermost inter-task loop results in an increment along the k^{th} placement dimension.

$$\{\forall I \in D | I + \mathbb{1}_d \in D : Pl(I + \mathbb{1}_d)_k = Pl(I)_k + 1\} \tag{12}$$

What Equation (12) states is that if a value of I in the iteration domain is placed at the end of the k^{th} dimension of the processor grid, and the next value along the innermost task loop dimension is placed at the beginning of that same dimension of the processor grid, the neighboring property is preserved along the (wrap-around) edge of the k^{th} dimension of a toric mesh.

3) Hexagonal: The interconnection network connectivity available in traditional meshes goes along the canonical directions ("North-South-East-West" in the 2-dimensional case). We can express a family of regular n-dimensional interconnection networks that have a different connectivity by intersecting a processor grid of more than n dimensions with hyperplanes. Hyperplanes are materialized by equalities in the polyhedron that defines the processor grid space.

Let us illustrate this with Kung and Leiserson's [12] two-dimensional hexagonal interconnect example, which can be represented as the intersection of a three-dimensional grid with the hyperplane defined by

$$Pl_0 - Pl_1 - Pl_2 = 0$$

The Kung-Leiserson interconnect is represented on Figure 2, and its corresponding representation as a polyhedral processor grid space is given by Figure 3. The intersection between the hyperplane and the grid is represented as a transparent surface (light red), while the points of the processor grid belonging to that hyperplane are circled in bold (red). On

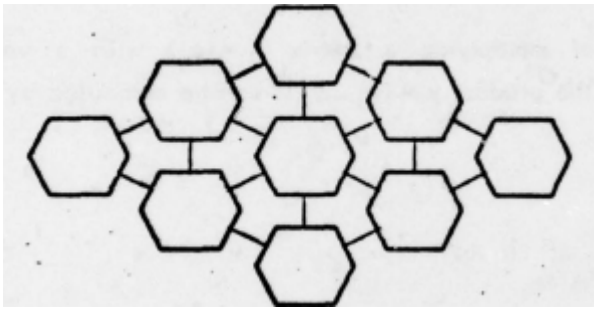

Fig. 2. Kung and Leiserson's hexagonal interconnect example

a mesh network, the definition of a neighbor is that the distance between processing element at coordinate $x \in \mathbb{N}^n$ is neighboring processing element at coordinate $y \in \mathbb{N}^n$ if and only if $||x - y|| \leq 1$. On the hexagonal network represented in Figure 3, the neighboring condition is instead $||x - y|| \leq 2$.

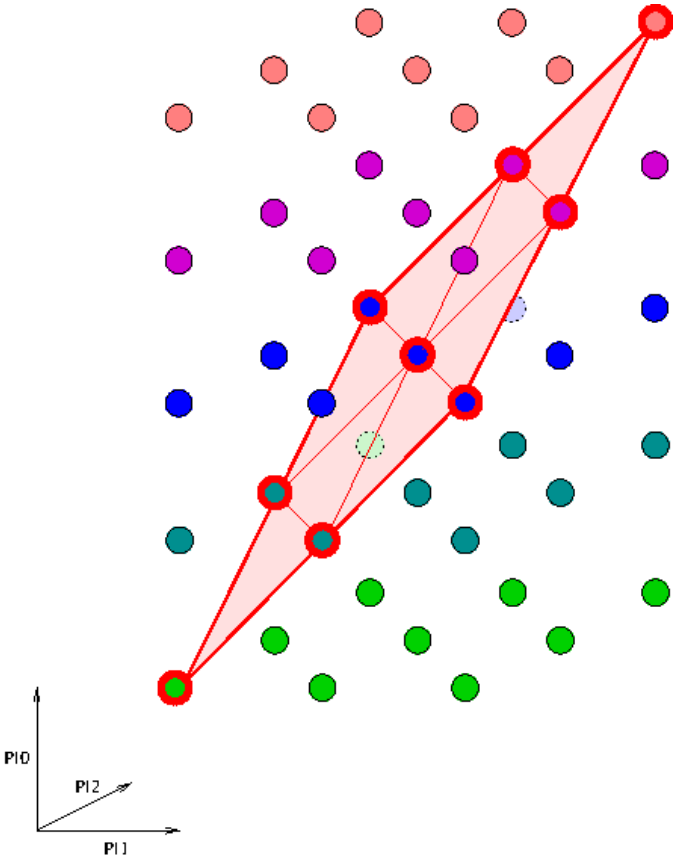

Fig. 3. Kung and Leiserson's hexagonal interconnect represented as a subspace of a 3-d processor grid

An octogonal two-dimensional interconnect can be represented by a two-dimensional grid associated with the neighboring condition $||x - y|| \leq 2$.

Note that the technique presented here does not focus on optimizing which neighbor-to-neighbor link should be used to perform the data transfer. This problem is left to a specific communication optimization pass.

B. Hierarchical

The way the presented broadcast elimination technique best applies to computer architectures with hierarchical parallelism depends upon their communication capabilities. Let us consider, without loss of generality, that there is an "outer" level of parallelism, materialized by "outer" processing elements (PEs), which contain "inner" PEs. Examples of such architectures are plentiful, for instance: x86 multi-cores as outer, their SIMD (Single Instruction, Mulitple Data) lanes as inner, or GPU (Graphics Processing Unit) accelerator cards as outer and their symmetric multiprocessor being inner.

1) Fine-grain inter-outer-PE communication: Some architectures offer fine-grain communication among inner PEs of neighboring outer PEs, as illustrated in Figure 4. We can view these as extending the spatial network that connects inner PEs through outer PE boundaries. These can be tackled as a "sea of inner processing elements", by flattening the

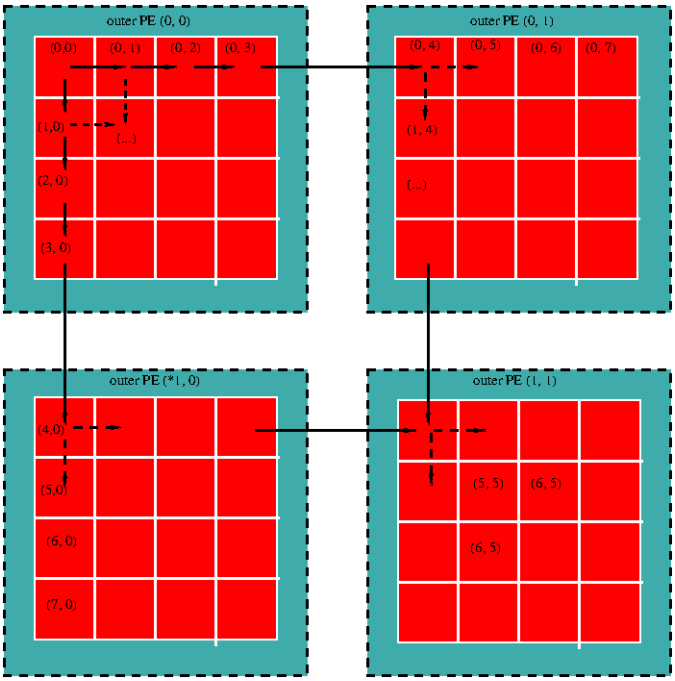

Fig. 4. Flattened view of a 2-dimensional PE grid with fine-grained inter-outer-PE capability

placement function along the processor dimensions where such fine-grained communications are possible between outer PEs. Let us denote $Pl_k^o(I)$ any such outer-PE dimension, and $Pl_k^i(I)$ the corresponding inner-PE dimension. Also, let S_k be the number of inner PEs along dimension k. Flattening is expressed by considering a new, one-dimensional placement function $Pl_k^f(I)$ defined as:

$$Pl_k^f(I) = S_k Pl_k^o(I) + Pl_k^i(I) \tag{13}$$

Our algorithm is then simply applied to the flattened grid, in which the dimensions allowing fine-grained inter-outer-PE communications are flattened.

2) Coarse-grain inter-outer-PE communication: The other case is when inter-outer-PE communication is optimized for coarse grain. The method presented here can still be used to eliminate broadcasts along any dimension of the outer and the inner grid. In this case, broadcast elimination at the outer PEs result in the transfer of chunks of data that are reused between neighboring outer PEs at once. This requires fewer synchronization than the fine-grain case above, but presumably results in higher latency, since the receiving outer PE has to wait for more producer-side inner PEs to produce their output data before it can start feeding its own inner PEs.

VII. Conclusion

We presented a family of affine scheduling algorithms for removing broadcast operations on a multi-dimensional grid of PEs, given an existing schedule and placement function. The algorithms apply to spatial architectures that can benefit from turning broadcast communications into neighboring ones.

They define a schedule such that reuse of data across the PE grid happens at consecutive (fine or coarse) time steps and between neighboring PEs. On architectures for which data transfers must be expressed directly (such as through a DMA programming API), a complementary optimization is required, which promotes the remote-to-local memory transfers to local-to-neighboring-local transfers. This complementary transformation is out of the scope of this paper. However, here we gave elements of evidence that the presented technique can dramatically reduce remote-to-local memory traffic, and hence potentially avoid bandwidth contention issues, as well as improve power consumption associated with data transfers.

REFERENCES

[1] Muthu Baskaran, Nicolas Vasilache, Benoît Meister, and Richard Lethin. Automatic generation of gpu-accelerated code for seismic stencil applications. In *Eighty-First Annual Meeting of Society of Exploration Geophysicists, SEG 2011*, September 2011.

[2] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, Tucson, Arizona, June 2008.

[3] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[5] Guy Even. The retiming lemma: A simple proof and applications. *The VLSI journal*, pages pp.123–137, 1995.

[6] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[8] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies, 2006.

[9] Martin Gusev and David Evans. Elimination of the computational broadcast: an application to qr decomposition. Technical Report 676, Parallel Algorithm Research Centre, Loughborough University of Technology, 1992.

[10] Michael James et al. Physical mapping of neural networks on a wafer-scale deep learning accelerator. In *Proceedings of the 2020 International Symposium on Physical Design*, 2020.

[11] N. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th Annual International Symposium on Computer Architecture*, 2017.

[12] H. T. Kung and Charles E. Leiserson. Systolic arrays for VLSI. Technical Report CMU-CS-79-103, Department of Computer Science, Carnegie-Mellon University, 1979.

[13] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 23–36, 1981.

[14] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. Supercomputing*, 21(1):37–76, 2002.

[15] V. Loechner and Catherine Mongenet. Communication optimization for affine recurrence equations using broadcast and locality. *International Journal of Parallel Programming*, 28(1):47–102, 2000.

[16] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream compiler. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[17] Alexis Platonoff. Automatic data distribution for massively parallel computers. Technical report, Centre de Recherche en Informatique / École Normale Supérieure, 1995.

[18] Patrice Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, October 1989.

[19] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[20] LTD Tilera. Tile processor architecture technology brief, 2007.

[21] Hongbin Zheng et al. Optimizing memory-access patterns for deep learning accelerators. Technical Report arXiv:2002.12798, Amazon Web Services, 2020.