More Data Locality for Static Control Programs on NUMA Architectures

Adilla Susungi, Albert Cohen, Claude Tadonki

More Data Locality for Static Control Programs on NUMA Architectures

Adilla Susungi
MINES ParisTech, PSL Research University

Albert Cohen
INRIA and DI, École Normale Supérieure

Claude Tadonki
MINES ParisTech, PSL Research University

ABSTRACT

The polyhedral model is powerful for analyzing and transforming static control programs, hence its intensive use for the optimization of data locality and automatic parallelization. Affine transformations excel at modeling control flow, to promote data reuse and to expose parallelism. The approach has also successfully been applied to the optimization of memory accesses (array expansion and contraction), although the available tools in the area are not as mature. Yet data locality also depends on other parameters such as data layout and data placement relatively to the memory hierarchy; these include spatial locality in cache lines and scalability on NUMA systems. This paper presents Ivie, a parallel intermediate language which complements affine transformations implemented in state-of-the-art polyhedral compilers and supports spatial and NUMA-aware data locality optimizations. We validate the design of the intermediate language on representative benchmarks.

Keywords

data locality, parallel intermediate language, NUMA systems, data layout

1. INTRODUCTION

Writing a program with good data locality involves multiple expertises, from the application domain to the computer architecture. It is also an intrinsically non-portable approach. This task must therefore be left to automatic tools capable of finding the appropriate code transformations. Loop transformations (e.g., tiling, fusion or interchange), layout transformations and data placement are examples of techniques involved in temporal and spatial locality optimization.

The powerful abstraction provided by the polyhedral model has led to its intensive use for analyzing and performing many of these transformations, targeting regions of the control flow that fit the model constraints (SCoPs). Today, polyhedral tools are capable of producing highly optimized parallel code for multicore CPUs [19], distributed systems [18], GPUs [39] or FPGAs [36].

Unfortunately, data layout transformations are not always taken into account as a first-class citizen in the design and implementation of such tools. Furthermore, an important category of multicore CPUs has been overlooked by most polyhedral compilation studies: Non-Uniform Memory Access (NUMA) systems. These large scale systems organize the physically shared memory across several nodes connected through cache-coherent high-performance links. With such a design, threads either access the memory resources located on the same node as the core executing the thread—the local node—or on a different node—a remote node. Consequently, uncontrolled data placement may yield traffic contention when all threads are accessing the same memory bank. Moreover, accessing data on remote nodes may increase memory latency. NUMA systems were initially introduced as a cure to the scalability issues of symmetric multiprocessors with Uniform Memory Access (UMA), yet ironically, NUMA-unaware programs running on NUMA platforms tend to not benefit from the additional computational power and memory bandwidth offered by the multiple nodes of the system. NUMA-awareness is commonly introduced at run time using tools such as `numactl` [7], but introducing NUMA-awareness to the application or better, to the compiled code automatically, provides much greater performance portability and transparency for the user.

The purpose of this paper is twofold:

- highlighting the benefits of integrating NUMA-awareness and layout transformations, more specifically transpositions, in polyhedral compilation;
- providing a concrete means for implementing these features in a polyhedral tool.

As a proof of concept, Pluto [19] is our demonstration framework but our work is applicable to other tools. It is probably best to consider integrating these features as an ad-hoc implementation in the considered polyhedral tool. Nevertheless, we choose to add such features by the means of a *parallel intermediate language* which design is currently in progress. This allows us to anticipate future extensions beyond static control region, as our aim for such an intermediate language is its integration in a general purpose compiler.

To model different loop- and data-centric optimizations, we implement our intermediate language as a meta-language. Meta-programming facilitates the development of prototypes to study the impact of data placement and layout transformations on a given program. In turn, this study allows to

IMPACT '17 January 23, 2017, Stockholm, Sweden

ACM ISBN none.

DOI: none

deduce a general algorithm that can be implemented towards complete automation from a high-level programming language such as OpenMP C.

Our contributions are as follows:

- We present Ivie, a parallel intermediate language (IL) focusing on the abstraction of array layout and mapping and on the ordering of the operations applied to these arrays. The IL decouples the expression of data transposition, implicit transposition through access functions, and placement on NUMA nodes.
- We provide a prototype compilation flow where Pluto is interfaced with a downstream automatic transformation step based on the IL.
- We present experiments on several, automatically parallelized PolyBench programs [8]. We study the effect of controlling the data placement on NUMA nodes and of different ways to implement array transpositions (explicitly or implicitly), considering both naively parallelized and tiled versions generated by Pluto.

This paper is structured as follows. The Ivie parallel intermediate language is presented in Section 2, the compilation flow for Pluto in Section 3 and experimental results in Section 4. Finally, in Sections 5, 6 and 7, we respectively discuss future work, related work and conclude.

2. IVIE, AN ARRAY-CENTRIC PARALLEL INTERMEDIATE LANGUAGE

We introduce the Ivie parallel intermediate language (IL), where arrays are considered as first-class data structures with decoupled data layout and NUMA allocation policy control. We choose a decoupled approach for manipulating arrays: the layout, NUMA mapping, and subscript expressions can be modified independently from one another. The language is also designed to ease the composition of array optimizations. Dedicated language constructs are provided for manipulating data layouts and memory allocation policies. And rather than explicit array subscripts, the language provides more abstracted iterators and additional constructs to manipulate iteration spaces—effectively making loop bounds implicit. Figure 1 describes the syntax the language. We will informally define each construct through small examples.

Let us first present the abstraction of access functions, then introduce the memory model and layout for the different types of arrays, and finally the NUMA mapping of these.

2.1 Access Function Abstraction

A nested loop is abstracted as follows:

```
# First kernel of Gemver
with i as piter:
  with j as siter:
    A[i][j] = k1(A[i][j], u1[i], v1[j], u2[i], v2[j])
```

First, we need to distinguish iterators for parallel loops from those for sequential loops. Hence, the `with` construct is used to declare an iterator and the `as piter` and `as siter` constructs are respectively used to qualify the declared iterator as used in a parallel and a sequential loop. Secondly, statements are abstracted as functions performing element-wise operations. That is, it takes as arguments and returns array elements only. The order in which arguments are listed corresponds to their order of appearance in the source program. Inductive definitions are made explicit by listing the target array element as the first one in the list.

Arrays have a dimension and a layout, set at declaration time. The layout of multi-dimensional may be transposed, i.e., the storage ordering of its elements in memory may follow a permutation of its dimensions.

2.2 Memory Model and Types of Arrays

Arrays may follow a *virtual* and a *physical* memory abstraction. The virtual memory abstraction is a single memory block whereas the physical memory abstraction may take the form of a collection of memory blocks distributed over different NUMA nodes. Virtual arrays are abstract views of physical arrays, derived from the latter through a specific declaration construct.

Physical arrays match the row-major layout of C arrays. Arrays of the source code processed by Pluto are mapped to physical arrays when generating the intermediate language from the Pluto output. Virtual arrays insert a layer of abstraction before physical indexing. For example, they may reorder the dimensions of the attached physical array so that indexes in the virtual array may implicitly translated into transposed index of the underlying physical array. The Pluto output is analyzed to detect all occurrences of transposed data accesses, and virtual arrays abstracting the associated dimension reorderings are declared accordingly. For instance, a column-major traversal of the form `A[j][i]` will lead, in Ivie, to the declaration of a virtual array `A_v` associated with the physical array `A`, such that the subscript expression `A_v[i][j]` will implicitly refer to `A[j][i]`. This form of array layout virtualization, and the distributed and replicated patterns over NUMA node, are both inspired from the alignment constructs in High Performance Fortran [2].

2.3 Basic Physical Array Declaration

There are two basic constructs for declaring a physical array:

- `array(`$\mathtt{N}_d$`, dtype, [`$\mathtt{size}_1$`, .., `$\mathtt{size}_{N_d}$`])`, where $\mathtt{N}_d$ is the number of dimensions, `dtype` is the data type and $\mathtt{size}_k$ is the size of dimension k;
- `replicate(name)`, where `name` is that of the array that must replicated. The array declared using this construct inherits the number of dimensions, the data type, the dimensions sizes and the data layout of the original array.

```
# Default declaration mode
A1 = array(2, double, [N,N])

# Declaration via replication of existing array
A2 = replicate(A1)
```

When converting from Pluto generated code, we only use the `array()` declaration. The `replicate()` construct is used for meta-programming.

2.4 Data Layout Management

The only layout transformation currently supported is the transposition. Transpositions can be performed using explicitly transposed copies or loop permutations. Loop permutations can be mainly handled through interchanging `with`

⟨*iter_type*⟩ ::= `piter` | `siter` | `cpiter`

⟨*phy_array_decl*⟩ ::= ⟨*identifier*⟩ `= array(` ⟨*integer*⟩ `,` ⟨*datatype*⟩ `, [` ⟨*size*⟩ (`,` ⟨*size*⟩)* `])`
| ⟨*identifier*⟩ `= replicate(` ⟨*identifier*⟩ `)`
| ⟨*identifier*⟩ `= transpose(` ⟨*identifier*⟩ `,` ⟨*integer*⟩ `,` ⟨*integer*⟩ `)`

⟨*virt_array_decl*⟩ ::= ⟨*identifier*⟩ `= vtranspose(` ⟨*identifier*⟩ `,` ⟨*integer*⟩ `,` ⟨*integer*⟩ `)`
| ⟨*identifier*⟩ `= select([` ⟨*condition*⟩ `,` ⟨*identifier*⟩ `] (, [` ⟨*condition*⟩ `,` ⟨*identifier*⟩ `])* )`

⟨*array_elt*⟩ ::= ⟨*identifier*⟩ `[` ⟨*subscript*⟩ `](` [⟨*subscript*⟩ `])*`

⟨*statement*⟩ ::= ⟨*array_elt*⟩ `=` ⟨*func_name*⟩ `(` ⟨*array_elt*⟩ (`,` ⟨*array_elt*⟩)* `)`

⟨*compound*⟩ ::= `with` ⟨*iterator*⟩ `as` ⟨*iter_type*⟩ `:` (⟨*compound*⟩ | ⟨*statement*⟩)

Figure 1: Ivie IL main syntax

constructs. In addition, we need other constructs to combine them with the manipulation of access functions and perform explicitly transposed copies.

To control which dimensions are the subject of a permutation, array dimensions are ranked in a numerical order with respect to their depth; an N-dimensional array has its dimensions ranked from 1 to N from the outermost to the innermost dimension. Based on this principle, there are two types of transposition constructs for each type of array:

- `transpose(name, rank`$_1$`, rank`$_2$`)` for transposing a physical *or* virtual array into a *physical* array. This also serves as a physical array declaration;
- `vtranspose(name, rank`$_1$`, rank`$_2$`)` for transposing a physical *or* virtual array into a *virtual* array.

In both cases, `name` is the name of the array to be transposed, and `rank`$_1$ and `rank`$_2$ are ranks denoting the dimensions to be permuted.

Here is an example of the transposition of a 2-dimensional array A into a physical array B:

```
# Transposition stored in a physical array
B = transpose(A, 1, 2)
```

We specify ranks `1` and `2` to indicate the corresponding dimensions to be permuted. The order of specification for `rank`$_1$ and `rank`$_2$ is interchangeable, meaning that `transpose(name, rank`$_1$`, rank`$_2$`)` is interpreted exactly the same way as `transpose(name, rank`$_2$`, rank`$_1$`)`.

This was trivial example. Now, let us assume that array A is a 4-dimensional array for which we want to permute ranks 1 with 3 and 2 with 4. The principle remains the same but requires successive calls to `transpose()` as in the following sample:

```
# Permuting several dimensions of the same array
A2 = transpose(A, 1, 3)
B  = transpose(A2, 2, 4)
```

Note that if we specify `B = transpose(A, 2, 4)` instead of `B = transpose(A2, 2, 4)`, the resulting array corresponds to array A with only ranks 1 and 3 permuted. Therefore, when permuting several dimensions of the same array, we need to make sure that at each step, the array of origin stores the latest accumulation of permutations. As memory management is made explicit, it is forbidden to encode such accumulation of permutations as `B = transpose(transpose, 2, 4), 1, 3)`.

These principles also apply to `vtranspose()`.

While their syntactic use is similar, using either `transpose()` or `vtranspose()` have different impact on the generated code. Using `transpose()` generates an explicitly transposed copy into a new array and its corresponding function accesses are modified accordingly. Generated code for accumulated permutations is to be optimized and the loops performing the copies should be properly scheduled. On the other hand, as `vtranspose()` creates a virtual array that will not appear in the generated code, using it allows us to abstract and modify the access function of an array.

2.5 Data Placement for NUMA

We currently handle two types of placement policies: interleaved allocation and allocation on a specific nodes. Data placement can only be applied to physical arrays as they are stored in the NUMA memory model. At this stage of the language design, these directly correspond to their respective Linux NUMA API functions [6] (`numa_alloc_interleaved()` and `numa_alloc_onnode()`).

2.5.1 *Interleaving*

Interleaved allocation of an array consists in mapping memory blocks of size N_b across a set of NUMA nodes in a round-robin fashion. The `numa_alloc_interleaved()` function performs this on all nodes available using page sizes only (4096 bytes). In addition, we would like to support any size multiple[1] of a page size as we consider implementing an extended version of `numa_alloc_interleaved()`. Therefore, considering that the minimum block granularity required is 4096 bytes, we introduce:

- `map_interleaved(granul)`, where `granul` > 0, so that $N_b = 4096 \times$ `granul`. The following code sample is an example:

```
# Distribution of each page
A.map_interleaved(1)

# Distribution by pairs of 2 pages
B.map_interleaved(2)
```

2.5.2 *Allocation on Specific Node*

Allocating a physical array on a specific node is done using the construct:

[1]Data mapping in the memory necessarily involves page sizes; even when wanting to map a single element (e.g, 8 bytes), the entire page size containing the element will be mapped. Hence, we can only handle multiple of page sizes.

- `map_onnode(id)` where `id` is a node id, as shown in the following sample:

```
# Allocation of A on node 1
A.map_onnode(1)

# Allocation of B on node 2
B.map_onnode(2)
```

2.5.3 Data Replication on Several Nodes

Data replication on several nodes requires *(i)* using `replicate()` and `onnode()`, *(ii)* introducing `as cpiter`, a new category of iterators and *(iii)* virtual arrays declared using a special construct:

- `select([cond`$_1$`, name`$_1$`], ..., [cond`$_N$`, name`$_N$`])`

where `[cond`$_x$`, name`$_x$`]` is a pair associating a condition `cond`$_x$ to an array `name`$_x$. The number of arguments is equal to the number of NUMA nodes considered. Conditions are specified with respect to the NUMA system's topology and additional thread scheduling.

Supposing that we need to replicate array A on 2 NUMA nodes, Listing 1 provides a full example of how to do so. Listing 2 is an example of generated code.

This example shows the complementary usage of virtual arrays besides abstracting function accesses. As one purpose of replicating the same array on several node is to avoid threads performing remote accesses, in a parallel loop, we need to modify the control flow so that each thread accesses the closest copy provided that thread migration is disabled. Using `select()` allows us to concisely abstract such modification. In the condition, `{it}` denotes an iterator that is the same across all conditions of the same `select()`. It may also denote the retrieval of a thread ID using an API function. In order to identify to which iterator `{it}` actually corresponds in the loop of interest, `as cpiter` is used. Nevertheless, we look forward to provide a more robust abstraction than `{it}` to be able to handle a wider range of applications. Several virtual arrays declared using `select()` within the same statement must either have the same set of conditions, or a least a subset of the greatest set of conditions that exists.

Remarks. In theory, any `piter` can be transformed into a `cpiter`, meaning that any parallel dimension in the same nested loop is eligible for such an iterator characterization. Furthermore, due to the composability of array declarations, an array associated to a condition can also be a virtual array declared using `select()`. In practice, these would definitely require optimizations but in the case of SCoPs, this is unlikely to occur. Therefore, we omit the extended syntax that allows us to match a `cpiter` to virtual array and we assume only one `cpiter` per loop.

Preventing threads from performing remote accesses can be done in different ways. We currently limit ourselves to the method shown in Listing 2, despite not being the most optimal. Other more efficient methods are considered as future work.

2.6 Implementation and Code Generation

Ivie IL perfectly fits into the syntax of Python. Thus, it is implemented similarly to a domain-specific language embedded into Python for quick usage as a meta-programming language. Even though our implementation is close to deep embedding, as an *intermediate* language, we have a slightly different approach.

```
A  = array(2, double, [N,N])
A2 = replicate(A)

A.map_onnode(0)
A2.map_onnode(1)

a = select([({it} % 2) == 0, A], [({it} % 2) != 0, A2])

with i as cpiter:
  with j as siter:
    ... = f(a[i][j])
```

Listing 1: Allocation of A and A2 respectively on node 0 and node 1 and consequence on the access function.

```
#pragma omp parallel for private(j) schedule(static,1)
for (i = 0; i < N; i++) {
  if ((i % 2) == 0) {
    for (j = 0; j < N; j++)
      ... = A[i][j];
  }
  if ((i % 2) != 0) {
    for (j = 0; j < N; j++)
      ... = A2[i][j];
  }
}
```

Listing 2: Example of code generated from the Ivie code in Listing 1

We reuse the existing expression tree, that is, the C abstract syntax tree (AST) already provided after parsing the source code using pycparser [9]. More precisely, we *parse* the Ivie program to generate its corresponding Python AST using RedBaron [11], a parser and full syntax tree (FST) generator for Python programs. This allows us to perform automatic IL transformations in the near future. When the Ivie program is transformed and ready for code generation, its FST is analyzed for extracting any information concerning arrays. Then, the C AST is modified with respect to the extracted information. Finally, we generate the resulting OpenMP C code.

Remark. As stated in Section 2.1, the order of appearance of arguments corresponds to the order in which array elements appear when traversing the C AST. In an inductive definition, the target array is positioned as the first argument. Consequently, modifying an argument in Ivie code results into the modification of the array element whose position matches in the C AST.

RedBaron's functionalities ease our decoupling principle. Indeed, we do not need to explicitly traverse the Python AST to extract information; we can directly *find* different categories of nodes using pattern matching. Thus, when wanting to extract any information on array allocation, we just need to find the list of nodes containing the string "map_". We then identify to which array name the allocation policy is associated and we store it in the data structure corresponding to the concerned array.

3. A MODIFIED PLUTO TOOL FLOW

Our IL can be integrated into the flow of Pluto to enable NUMA-awareness and additional layout transformations. We simply need to generate an Ivie program from a Pluto-generated code, exporting the relevant semantical properties (starting with parallelism) from bands of per-

mutable dimensions exposed by the Pluto algorithm. To simplify this process, we delimit array declarations and SCoPs using pragmas. The tool flow is as follows (Pluto default steps in italic):

1. *Extract polyhedral representation using Clan.*
2. *Perform optimizations and parallelization.*
3. *Generate C code using CLooG.*
4. *Apply GCC preprocessing on the CLooG-generated code to replace each* `#define` *by their actual statements.*
5. Generate the Ivie program:
   A. collect array declarations;
   B. parse the preprocessed code;
      and for each dimension in each band,
      i. if the dimension is marked parallel, characterize the Ivie iterator as `piter`,
      ii. if not, characterize it as `siter`;
   C. print output Ivie file.
6. [Currently handled by hand] Meta-program the desired Ivie code transformation.
7. Parse the Ivie program and generate its Python AST using RedBaron.
8. Parse the OpenMP C program and generate its C AST using pycparser.
9. From the Python AST, collect all information concerning arrays and store them in data structures, creating sets of physical arrays and virtual arrays.
7. Proceed with C AST modification, using the Pluto-generated code and the transformed Ivie meta-program.
8. Generate code from modified C AST.

We currently express Ivie code transformations through manual (Ivie) meta-programming. This should be automated when the appropriate heuristics will be stabilized. The present study can be seen as an important step towards the design and implementation of such automated optimizations.

4. EXPERIMENTAL RESULTS

The purpose of this section is to present the effects of additional NUMA allocation and transposition in different application scenarios. There is a clear separation between the optimizations handled by Pluto and those handled by Ivie IL. Pluto is in charge of all control flow optimizations and parallelism extraction, whereas our post-pass implements data placement on NUMA systems as well as the actual transpositions. Pluto outputs may be complex; hence we handle affordable cases. The automated search for optimal solutions involving, for instance, the time spent in any additional data copying, or coupling NUMA decisions which additional runtime specifications such as thread binding, is part of future work.

We present case studies of several PolyBench [8] programs: Gemver, Gesummv, Gemm and Covariance. The minimal set of Pluto options are tiling for L1 cache, parallelism and vectorization. All programs are compiled with `gcc -O3 -march=native` which enables vectorization by default. We execute them on a 2 sockets NUMA system with 36 Intel Xeon cores E5-2697 v4 (Broadwell) at 2.30 GHz distributed across 4 nodes (9 cores per node, L1 cache: 32K).

NUMA optimizations involve interleaved allocation and data replications. We use simple guidelines to decide which policy to choose:

- An array is replicated on all nodes if each thread perform *read-only* accesses on the *entire* array. As replicating written arrays would require additional heuristics to ensure data coherence, we do not yet handle such cases.
- The remainder may be interleaved on all nodes, especially multi-dimensional arrays, to reduce traffic contention.

Following this rule, Gemver appears to be the only program in which we apply additional data replications.

Transpositions are performed at initialization time using indexes permutation only. Although our framework supports transpositions at well-defined points in the control flow, no such additional data copying was attempted in the current set of experiments.

Some examples of meta-programs are provided in Appendix.

Gemver. We compare different Pluto-generated versions of Gemver:

- The default output of Pluto;
- The addition of NUMA placement only, using replication and interleaved allocation;
- The addition of transpositions only;
- The addition of combined NUMA placement and transposition.

Moreover, we consider two Pluto outputs as our baselines: the first is generated using the *no fuse* heuristic and the second, the *smart fuse* heuristic. We do not consider the *max fuse* option as it is not suitable for Gemver. Figure 2 shows the speedup of each program with respect to the default Pluto output considered.

The *smart fuse* version scales better that the *no fuse* version. Indeed, the fusion of the two first loops favours cache reuse. However, compared with a naive parallel version of Gemver, the two Pluto outputs fare no better, meaning that optimizing temporal locality only is not sufficient. As 10 cores is the threshold from which NUMA effects appear, we can also see in Figure 2 that they poorly scale with 16 and 36 cores.

According to the aforementioned guidelines, we applied the exact same NUMA placement in both outputs. While this solution improves both, *no fuse* provides the best performance with 36 cores. When using interleaved allocation for an array, the different threads accessing it must perform row-major accesses to preserve node locality as much as possible. This is the case with *no fuse*. With *smart fuse*, the first loop is permuted in order to perform the fusion legally. Despite enhancing cache reuse, as column-major accesses are still performed, some node locality is lost.

Thread binding using `OMP_PROC_BIND` seem not to significantly improve performance. It may even lower the speedup. We noticed that when performing the same experiments with NUMA placement based on replications only, the positive effects of thread binding were much more noticeable despite less absolute speedup. Interleaved allocation therefore seem to inhibit the effects of thread binding as it may reduce node locality per threads; in this case, thread migration may operate as a counterbalance.

On the other hand, layout transformations are better suited to *smart fuse* as data reuse is much more enhanced. Furthermore, this allows threads to perform row-major accesses with respect to interleaved mapping. Yet such a systematic transformation does not align well with the schedule of the

no fuse version, hence the degraded performance. Therefore, at this level of optimizations, the best version of Gemver appears to involve smart fuse, NUMA placement and transposition.

Additional data replications using `memcpy` add 0.3 ms to all execution instances of versions with NUMA placement. They therefore have a negligible impact on the performance observed.

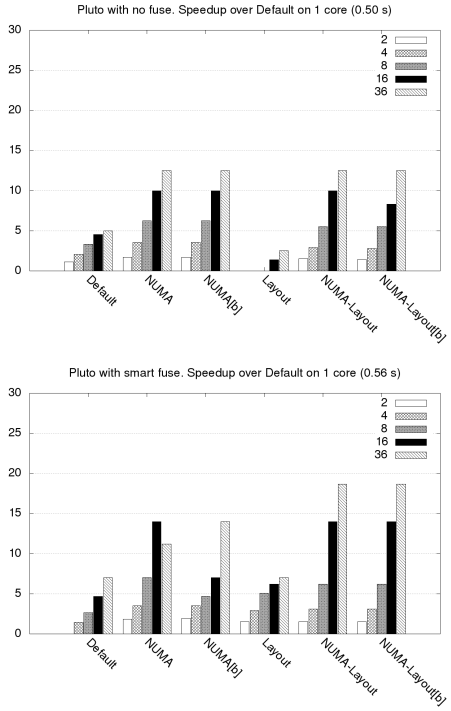

Figure 2: Speedups for different versions of Gemver. Execution instances with thread binding marked with `[b]`

.

Gesummv. Gesummv with *no fuse* and *max fuse* are our Pluto baselines[2]. Results are depicted in Figure 3. Similarly to the case of Gemver, optimized temporal locality alone does not provide better performance than the naive parallel version for Pluto-generated codes. These also scale poorly without NUMA-aware placement. We applied interleaved allocation, which brings a 3× speedup on 16 cores and 4× speedup on all 36 cores. As we observed that thread binding may not match with interleaved allocations, we consider exploring the effects of changing the granularity of interleaving for further improvements.

Covariance. In this case, Pluto delivers much better performance than the naive parallel version. We compare versions with NUMA placement only (interleaved data allocation), transposition as shown in Figure 4, and both together. Figure 5 shows that the main improvement on the naive parallel version results from the transposition. However, applying the same transposition to the Pluto output considerably hurt the program's performance, similarly to what we observe for Gemver with no fuse. NUMA allocations have little positive impact on the naive version and none on the Pluto output, given that temporal locality already optimizes cache usage.

[2] *No fuse* and *smart fuse* heuristics result into the same generated code.

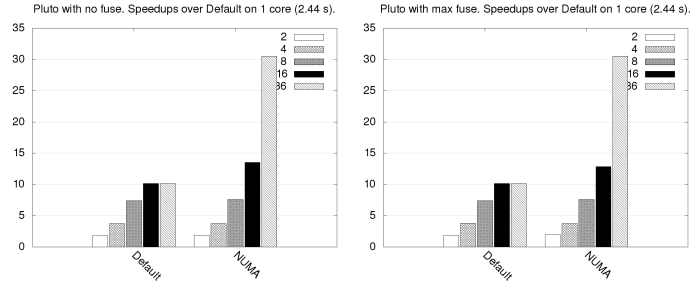

Figure 3: Speedups for different versions of Gesummv

```
// Initializing the transposition
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    data[j][i] = ((DATA_TYPE) i*j) / M;

// Computation
#pragma omp parallel for private(i)
for (j = 0; j < _PB_M; j++) {
  mean[j] = SCALAR_VAL(0.0);
  for (i = 0; i < _PB_N; i++)
    mean[j] += data[j][i];
  mean[j] /= float_n;
}
#pragma omp parallel for private(j)
for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_M; j++)
    data[i][j] -= mean[i];

#pragma omp parallel for private(j,k)
for (i = 0; i < _PB_M; i++)
  for (j = i; j < _PB_M; j++) {
    cov[i][j] = SCALAR_VAL(0.0);
    for (k = 0; k < _PB_N; k++)
      cov[i][j] += data[i][k] * data[j][k];
    cov[i][j] /= (float_n - SCALAR_VAL(1.0));
    cov[j][i] = cov[i][j];
  }
```

Figure 4: Covariance with transposition of array `data`

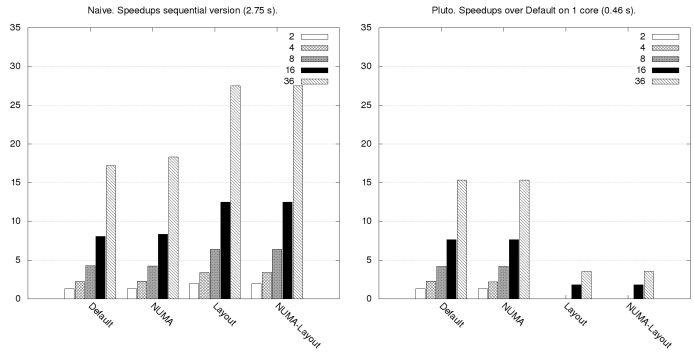

Figure 5: Speedups for different versions of Covariance

Gemm. As shown in Figure 6, all versions scale very well, but additional locality optimization and vectorization in Pluto-based outputs considerably improve performance. We measure the execution time of multiple untiled parallel versions including two methods for eliminating column-major accesses. The first version eliminates such accesses

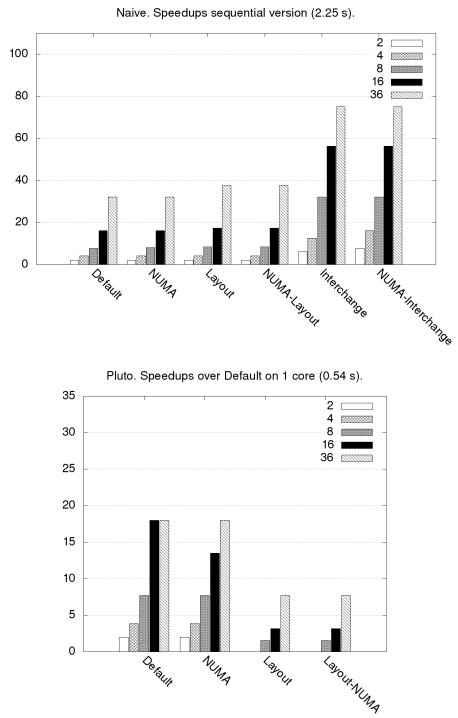

Figure 6: Speedups for different versions of Gemm

through data transposition, and the second version is obtained through loop interchange (ikj). Loop interchange is also the transformation that Pluto performs, in addition to tiling. On this machine, loop interchange seems more appropriate than data transposition on the naive parallel version. We reproduced these experiments on two other machines: a 4 core Intel Core i7-4910MQ CPU (Haswell) at 2.90GHz and a 16 core Intel Xeon CPU E5-2660 (Sandy Bridge) at 2.20GHz. While we observed the same tendency on the Haswell with or without the `-O3` option, we noticed the opposite on the Sandybridge when disabling this option. This is probably due to the lower AVX computation throughput compared to memory bandwidth: the program (with these optimizations) is definitely compute-bound on Sandybridge and NUMA optimizations have little impact.

These results show that, on one hand, bandwidth-bound programs that cannot be improved using Pluto's default heuristics (Gemver and Gesummv) do benefit from heuristics on NUMA placement and transpositions. On the other hand, programs already benefiting a lot from Pluto's heuristics do not require additional NUMA placement as most data accesses hit the cache. Moreover, in some cases and depending on the machine, it seems wiser to rely on loop scheduling transformations rather than data layout transformations.

These case studies show the advantages of complementing polyhedral tools with placement heuristics on NUMA nodes and transposition. NUMA placement tend to improve a program's scalability, especially from the threshold from which the NUMA effect appears. On the other hand, transposition helps improving the absolute speed-up. Combining both is an interesting alternative provided that further optimizations are performed. Indeed, control flow modifications introduced to incorporate more optimizations opened the door for additional loop optimizations that we were not yet able to implement.

Introducing Ivie IL in Pluto allows widening the range of options for optimizing SCoPs, especially on NUMA systems. Through static analyses, optimization decisions could be automatically provided. For NUMA optimizations, this could reduce the need to handle all heuristics at execution time.

5. FUTURE WORK

Our first experiments call for deeper investigation of the interplay of data layout and schedule transformations, and for the design and implementation of the associated heuristics.

Control flow optimizations. For the moment, all control flow optimizations are left to Pluto. For greater impact, we need to integrate complementary control flow optimizations on the IL itself. This has several advantages:

- Performing layout transformations may open the door for further loop transformation such as fusion. On the other hand, data replications as currently performed definitely require some form of hoisting. More generally, the methods for handling replicated data may generate as many loops as there are replicated arrays and assign each loop to a thread group. Being able to manipulate the control flow will allow us to generate more efficient programs.

- Concerning replications again, the method involving complementary hoisting may only be implemented in a polyhedral framework if the condition is affine. Yet replication conditions may depend on a built-in function (e.g., to retrieve the thread id, using a parallel programming API). Such transformations are out of the model's scope, and are typically not supported by compilers either. Thanks to the semantics of our IL, we will be able to handle such optimizations.

More expressiveness for loop iterators. At this stage of design, we distinguish iterators that are used in a parallel or a sequential loop. Such distinction is based on explicit parallelism exposed in the source code (with pragmas for instance). However, when dealing with Pluto's own internal tree representation(s), some dimensions may be parallel yet not *exploited as a parallel loop*, and some bands of dimensions may carry permutability properties. We need to carry these properties along the IL, to preserve the capability to implement loop permutations or any other loop transformations.

Implementation. For more complex code transformations, a deeper embedding into Python may be considered, instead of directly patching the existing AST. Indeed, modifications such as generating multiple loops, each one assigned to a thread group, may motivate the reconstruction of a complete tree. Maintenance and greater flexibility of the design are other motivations for such a refactoring.

Optimizations. Another aspect that must be taken into account is the time spent in data copying when replicating over several nodes. This depends on the machine and on the way arrays are allocated. The same applies to explicit transposition. We need to make sure that this does not overshadow the program's performance, and a model must be developed to better assess the profitability of replication and transposition. Polyhedral analyses may be leveraged to do so. Provided a known number of threads and a static thread scheduling, we can determine the set of elements accessed per thread, using the size of their loop chunk. Then,

we can choose which NUMA policy can be applied, for instance, according to the guidelines listed at the beginning of Section 4. If interleaved allocation is chosen, such information can also help determining the interleaving granularity. As for transpositions, we can determine different possible schedules for explicit transpositions and their possible transformation into implicit ones. Finally, using iterative compilation and auto-tuning, we may determine the appropriate choices for a given architecture.

6. RELATED WORK

Data placement using parallel languages. Optimizations for NUMA systems is a broad research topic. We therefore limit ourselves to closerly related work in programming language design and implementation. One motivation for handling NUMA placement in an intermediate language is the lack of such support in existing parallel programming languages. HPF, X10 or Chapel provide constructs for data distribution across processes. Chapel also includes preliminary support for NUMA in its locale models; OpenMP only provides the `OMP_PROC_BIND` environment variable to disable thread migration; both mechanisms take action at execution time only. One exception is GNU UPC [1], a toolset providing a compilation and NUMA-aware execution environment for programs written in UPC. Very few languages allow data distribution on NUMA nodes. This task is generally performed with third party tools such as libnuma [6] or hwloc [3]. Some attempts to fill this gap have been proposed for OpenMP [17, 25, 33] and Intel TBB [32]. Yet none of these have been integrated into the official language specifications.

Languages and efficient code generation. Ivie is not to be considered, strictly speaking, as a new programming language. Nevertheless, being implemented as meta-language to model optimizations, it presents some similarity with several other languages summarized in Table 1. Most of the presented languages target GPUs, except Halide [38] and PolyMage [34] but these do not provide explicit constructs for NUMA placement. Vobla [15] and Loo.py [30] appear to be the only languages that provide explicit constructs for performing layout transformations. However Loo.py provides additional layout transformation such as padding. Ivie is the only language that is used post-polyhedral techniques; besides the non polyhedral Halide, PolyMage and Xfor are fully polyhedral-based and Loo.py has access to polyhedral optimization by means of the islpy library. As for Vobla and Pencil, the latter implements domain-specific transformations and compiles to the latter, Pencil serving as an intermediate language for Vobla to generate high performance GPU code using PPCG [39]. Other non polyhedral-related solutions have been proposed to take into account data layout transformations, such as Raja [10] and Kokkos [5]. These tools target both CPUs and GPUs, and the ability to apply specific layouts to both architectures is crucial. Furthermore, Kokkos proposes an interface with hwloc, mostly for retrieving information on the topology and performing thread binding.

The approach of combining the polyhedral representation with another intermediate representation has also been applied in the CHiLL compiler targeting both GPUs and multicore. Indeed, Zhang et al. [41] compose AST transformations to polyhedral transformation for optimizations such as loop unrolling or partial sum for higher-order stencils.

Parallel optimizations in the polyhedral model. Being able to handle parallel programs within the polyhedral model has been the subject of several investigations. Basically, three main approaches have been used. The first approach is to use the model as is; Basupalli et al [14] consider the parallel construct `omp for` as a program transformation that assigns new time-stamps to instances of the program and Darte et al. [23] propose a method for liveness analysis where conflicting variables are computed based on the notion of partial order and the happens-before relationship that can be computed with the ISCC calculator. The second approach is characterizing an analyzable subset for a specific parallel language; indeed, Yuki et al [40], Pellegrini et al. [35] and Cohen et al [22] respectively defined polyhedral subsets of X10, MPI and OpenStream [37]. The last approach is extending the model, which has been proposed by Chatarasi et al [20].

Parallel intermediate languages. Despite being coupled with the polyhedral model in this paper, we are not working towards a polyhedral-specific intermediate language. We rather aim at general-purpose compilation of parallel languages, following the steps of INSPIRE [26] or SPIRE-d [28]. INSPIRE is the intermediate representation of the Insieme compiler [4] and SPIRE is a methodology for introducing the semantics of parallel programs in existing intermediate representations; proofs of concepts for these have been demonstrated for compilers such as PIPS [27] or LLVM [29]. Such intermediate representations abstract the semantics of parallel programs, i.e. parallel constructs, barriers or communications, but none of them handle data placement or layout transformation.

Memory optimizations. As for optimizations targeting memory, tools such as Bee+Cl@k [12] or SMO [16] tackle the reduction of memory footprint through intra- or even inter-array memory reuse. Clauss and Meister [21] also perform layout transformation to enhance spatial locality but our work is much more similar to that of Lu et al. [31]; it focuses on layout transformations for locality on NUCA (Non-Uniform Cache Access) multiprocessors. In this work, the polyhedral model is first used to analyze indexes and array reference functions, then layout optimizations are applied.

7. CONCLUSION

We highlighted the benefit of introducing NUMA-awareness and additional layout transformations to a state-of-the-art polyhedral flow. In particular, we demonstrated the feasibility of a static placement heuristic matching the NUMA topology. We illustrated these results on a modified version of the automatic parallelizer and locality optimizer Pluto, integrating it with Ivie, a specifically-designed parallel intermediate language to model such layout and mapping transformations. In Ivie, arrays are first-class data structures on which operations such as data placement or transposition can be performed. Future work involves being able to integrate more closely with control flow optimizations, adding expressiveness for further abstracting a diverse range of loop iterators, and a designing a more robust implementation of the code generator supporting complex compositions of optimizations.

8. REFERENCES

[1] GNU UPC. http://www.gccupc.org.

[2] HPF: High Performance Fortran. http://hpff.rice.edu.

[3] hwloc. https://www.open-mpi.org/projects/hwloc/.

Table 1: Current position of Ivie towards several languages. (N/A: not applicable)

	Target	Constructs for layout management	Constructs for NUMA placement	Position towards polyhedral techniques
Ivie	CPU	Yes	Yes	Post-polyhedral
Halide [38]	CPU / GPU	No	No	Custom
PolyMage [34]	CPU / GPU	No	No	Fully polyhedral
Xfor [24]	CPU	No	No	Fully polyhedral
Vobla [15]	GPU	Yes	N/A	Pre-polyhedral
Pencil [13]	GPU	No	N/A	Pre-polyhedral
Loo.py [30]	GPU	Yes	N/A	Includes polyhedral

[4] Insieme compiler. www.insieme-compiler.org.

[5] Kokkos. https://github.com/kokkos.

[6] Libnuma: Linux NUMA API. http://man7.org/linux/man-pages/man3/numa.3.html.

[7] Numactl. https://linux.die.net/man/8/numactl.

[8] PolyBench: Polyhedral Benchmark Suite. https://sourceforge.net/projects/polybench/.

[9] pycparser. https://github.com/eliben/pycparser.

[10] Raja. http://software.llnl.gov/RAJA/_static/RAJAStatus-09.2014_LLNL-TR-661403.pdf.

[11] Redbaron. http://redbaron.readthedocs.io/en/latest/.

[12] C. Alias, F. Baray, and A. Darte. Bee+cl@k: An implementation of lattice-based array contraction in the source-to-source translator rose. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 73–82, New York, NY, USA, 2007. ACM.

[13] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, 2015.

[14] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, IWOMP'11, pages 37–53, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA: A Vehicle for Optimized Basic Linear Algebra. *SIGPLAN Notices*, 49(5):115–124, June 2014.

[16] S. G. Bhaskaracharya, U. Bondhugula, and A. Cohen. Smo: An integrated approach to intra-array and inter-array storage optimization. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 526–538, New York, NY, USA, 2016. ACM.

[17] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[18] U. 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, pages 33:1–33:12, New York, NY, USA, 2013. ACM.

[19] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[20] P. Chatarasi, J. Shirako, and V. Sarkar. Static Data Race Detection for SPMD Programs via an Extended Polyhedral Representation. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques*, IMPACT '16, 2016.

[21] P. Clauss and B. Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News*, 28(1):11–19, Mar. 2000.

[22] A. Cohen, A. Darte, and P. Feautrier. Static Analysis of OpenStream Programs. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques*, IMPACT '16, 2016.

[23] A. Darte, A. Isoard, and T. Yuki. Liveness Analysis in Explicitly Parallel Programs. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques*, IMPACT '16, 2016.

[24] I. Fassi and P. Clauss. Xfor: Filling the gap between automatic loop optimization and peak performance. In *Proceedings of the 2015 14th International Symposium on Parallel and Distributed Computing*, ISPDC '15, pages 100–109, Washington, DC, USA, 2015. IEEE Computer Society.

[25] L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling Locality-aware Computations in OpenMP. *Sci. Program.*, 18(3-4):169–181, Aug. 2010.

[26] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 7–18, Piscataway, NJ, USA, 2013. IEEE Press.

[27] D. Khaldi. *Automatic Resource-Constrained Static Task Parallelization*. PhD thesis, MINES ParisTech, PSL Research University, 2013.

[28] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt. SPIRE : A Methodology for Sequential to Parallel Intermediate Representation Extension. In *HiPEAC Computing Systems Week*, Paris, France, May 2013.

[29] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and

B. Chapman. LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.

[30] A. Klöckner. Loo.py: Transformation-based code generation for gpus and cpus. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 82:82–82:87, New York, NY, USA, 2014. ACM.

[31] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 348–357, Washington, DC, USA, 2009. IEEE Computer Society.

[32] Z. Majo and T. R. Gross. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 227–238, New York, NY, USA, 2015. ACM.

[33] A. Muddukrishna, P. A. Jonsson, and M. Brorsson. Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors. *Scientific Programming*, 2015, 2015.

[34] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[35] S. Pellegrini, T. Hoefler, and T. Fahringer. Exact dependence analysis for increased communication overlap. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 89–99, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] A. Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD thesis, Ecole Normale SupÃl'rieure de Lyon, September 2010.

[37] A. Pop and A. Cohen. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53:1–53:25, Jan. 2013.

[38] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[39] S. Verdooleage, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[40] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array Dataflow Analysis for Polyhedral X10 Programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 23–34, New York, NY, USA, 2013. ACM.

[41] H. Zhang, A. Venkat, P. Basu, and M. Hall. Combining Polyhedral and AST Transformations in CHiLL. In *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques*, IMPACT '16, 2016.

APPENDIX

A. GEMVER SMART/NO FUSE WITH NUMA PLACEMENT AND TRANSPOSITION

The following is Gemver *smart fuse* with NUMA placement and transposition meta-programmed in Ivie.

```
# Default declarations
A = array(2, DATA_TYPE, [n, n])
u1 = array(1, DATA_TYPE, [n])
v1 = array(1, DATA_TYPE, [n])
u2 = array(1, DATA_TYPE, [n])
v2 = array(1, DATA_TYPE, [n])
w = array(1, DATA_TYPE, [n])
x = array(1, DATA_TYPE, [n])
y = array(1, DATA_TYPE, [n])
z = array(1, DATA_TYPE, [n])

# Meta-programmed declarations
A_v = vtranspose(A, 1, 2)
u1_1 = replicate(u1)
u1_2 = replicate(u1)
u1_3 = replicate(u1)
u2_1 = replicate(u2)
u2_2 = replicate(u2)
u2_3 = replicate(u2)

A.map_interleaved(1)
w.map_interleaved(1)
x.map_interleaved(1)
z.map_interleaved(1)
u1.map_onnode(0)
u1_1.map_onnode(1)
u1_2.map_onnode(2)
u1_3.map_onnode(3)
u2.map_onnode(0)
u2_1.map_onnode(1)
u2_2.map_onnode(2)
u2_3.map_onnode(3)

# {it} denotes here the retrieval of thread IDs
u1_s = select([0 <= {it} <= 8, u1],
              [9 <= {it} <= 17, u1_1],
              [18 <= {it} <= 26, u1_2],
              [27 <= {it} <= 35, u1_3])

u2_s = select([0 <= {it} <= 8, u2],
              [9 <= {it} <= 17, u2_1],
              [18 <= {it} <= 26, u2_2],
              [27 <= {it} <= 35, u2_3])

# Transposed initialization of A using A_v
with i as siter:
  with j as siter:
    A_v[i][j] = init()
```

```
# ... other initializations

# Tile dimension with t2 for t5
# Tile dimension with t3 for t4
with t2 as cpiter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        A[t4][t5] = f3(A[t4][t5], u1_s[t4], v1[t5],
                       u2_s[t4], v2[t5])
        x[t5] = f3(x[t5], A[t4][t5], y[t4])

with t2 as piter:
  with t3 as siter:
    x[t3] = f5(x[t3], z[t3])

with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        w[t4] = f9(w[t4], A[t4][t5], x[t5])
```

Gemver *no fuse* with NUMA placement and transposition is almost the same code. The following code shows the differences.

```
# Tile dimension with t2 for t4
# Tile dimension with t3 for t5
with t2 as cpiter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        A[t4][t5] = f3(A[t4][t5], u1_s[t4], v1[t5],
                       u2_s[t4], v2[t5])

# Tile dimension with t2 for t5
# Tile dimension with t3 for t4
with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        x[t5] = f7(x[t5], A[t4][t5], y[t4])
```

B. GESUMMV MAX/NO FUSE WITH NUMA PLACEMENT

The following is the *no fuse* version.

```
# Default declarations
A = array(2, DATA_TYPE, [n, n])
B = array(2, DATA_TYPE, [n, n])
tmp = array(1, DATA_TYPE, [n])
x = array(1, DATA_TYPE, [n])
y = array(1, DATA_TYPE, [n])

# Meta-programmed mapping
A.map_interleaved(1)
B.map_interleaved(1)

with t2 as piter:
  with t3 as siter:
    y[t3] = f1()

with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        y[t4] = f5(B[t4][t5], x[t5], y[t4])

with t2 as piter:
  with t3 as siter:
    tmp[t3] = f7()

with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        tmp[t4] = f11(A[t4][t5], x[t5], tmp[t4])

with t2 as piter:
  with t3 as siter:
      y[t3] = f13(tmp[t3], y[t3])
```

The *max fuse* version has the same NUMA mapping but fuses all loops as follows:

```
with t1 as piter:
  with t4 as siter:
    y[t4] = f1()

  with t3 as siter:
    with t4 as siter:
      with t6 as siter:
        y[t4] = f4(B[t4][t6], x[t6], y[t4])

  with t4 as siter:
    tmp[t4] = f5()

  with t3 as siter:
    with t4 as siter:
      with t6 as siter:
        tmp[t4] = f8(A[t4][t6], x[t6], tmp[t4])

  with t4 as siter:
    y[t4] = f9(tmp[t4], y[t4])
```

C. PLUTO-GENERATED GEMM WITH TRANSPOSITION

```
# Default declarations
C = array(2, DATA_TYPE, [ni, nj])
A = array(2, DATA_TYPE, [ni, nk])
B = array(2, DATA_TYPE, [nk, nj])

# Meta-programmed declaration
B_v = vtranspose(B, 1, 2)

# Initializations
with i as siter:
  with j as siter:
    B_v[i][j] = init()
# ... other initializations

with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        C[t4][t5] = f3()

with t2 as piter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        with t7 as siter: # t7 is interchanged with t6
          with t6 as siter:
            C[t5][t6] = f9(C[t5][t6], A[t5][t7],
                           B[t6][t7])
```