More Data Locality for Static Control Programs on NUMA Architectures

Adilla Susungi[1], Albert Cohen[2], Claude Tadonki[1]

[1]MINES ParisTech, PSL Research University
[2]Inria and DI, Ecole Normale Supérieure

7th International Workshop on Polyhedral Compilation Techniques (IMPACT'17)
Stockholm, Sweden, January 23, 2017

Motivations

Data locality

- Interest in any kind of technique that can produce data locality
- Combining several types
  - Loop transformations
  - Layout transformations
  - Data placement on NUMA architectures

Motivations

Data locality

- Interest in any kind of technique that can produce data locality
- Combining several types
    - Loop transformations
    - Layout transformations
    - Data placement on NUMA architectures

Automatic polyhedral parallelizers

- Current tools do not consider the integration of control, data flow, memory mapping and placement optimizations

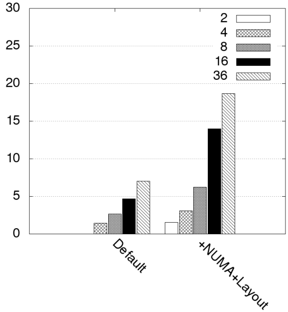
Example

- Pluto[1]: for multicore CPUs
  → Optimizations using loop transformations only

[1] 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.

Example

- Pluto[1]: for multicore CPUs
  → Optimizations using loop transformations only

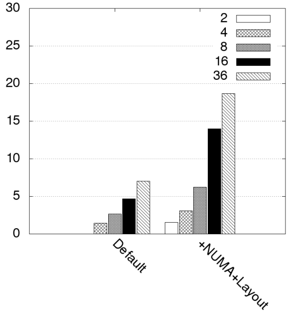


- 16 cores: 3x
- 36 cores: 2.6x

[1] 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.

Example

- Pluto[1]: for multicore CPUs
  → Optimizations using loop transformations only



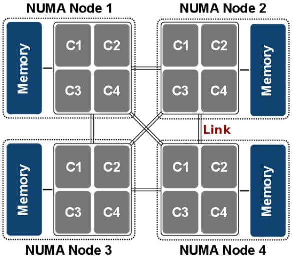
- 16 cores: 3x
- 36 cores: 2.6x

How to provide more data locality thanks to additional transpositions and NUMA placements?

[1] 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.

NUMA architectures

Traffic contention and remote accesses issues



Can be dealt with:

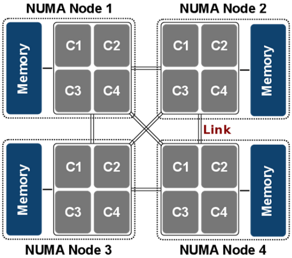
- At the programming level using an API (`Libnuma`, `hwloc`)
- Using extended programming languages[2]
- At execution time using environment variables (`GOMP_CPU_AFFINITY`, `KMP_AFFINITY`) or runtime solutions (e.g, MPC[3])

[2] 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.

[3] M. Pérache, H. Hourdren, R. Namyst. *MPC: A Unified Parallel Runtime for Clusters of NUMA Machines*. Euro-Par 2008.

NUMA architectures

Traffic contention and remote accesses issues



Can be dealt with:

- At the programming level using an API (`Libnuma`, `hwloc`)
- Using extended programming languages[2]
- At execution time using environment variables (`GOMP_CPU_AFFINITY`, `KMP_AFFINITY`) or runtime solutions (e.g, MPC[3])

What is the most convenient way to explore NUMA placement decisions at compile-time?

[2] 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.

[3] M. Pérache, H. Hourdren, R. Namyst. *MPC: A Unified Parallel Runtime for Clusters of NUMA Machines*. Euro-Par 2008.

Roadmap

Goals

1. Transpositions and NUMA placements in Pluto outputs for more locality
2. A convenient way to explore optimizations decisions at compile-time

Roadmap

Goals

1. **Transpositions and NUMA placements in Pluto outputs for more locality**
2. **A convenient way to explore optimizations decisions at compile-time**

Our solution

- **Proposing a parallel intermediate language: Ivie**
  - $\rightarrow$ Manipulate meta-programs for space exploration
  - $\rightarrow$ Makes prototyping easier than using unified polyhedral approach
  - $\rightarrow$ Future use beyond SCoPs
- **Prototyping an extension of Pluto tool flow involving the PIL**
  - $\rightarrow$ Case studies on PolyBench programs: Gemver, Gesummv, Covariance, Gemm

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

Design

- Declarative/functional

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

Design

- Declarative/functional
- Decoupled manipulation of array characteristics

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

Design

- Declarative/functional
- Decoupled manipulation of array characteristics
- Physical and virtual memory abstraction

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

Design

- Declarative/functional
- Decoupled manipulation of array characteristics
- Physical and virtual memory abstraction
- Meta-language embedded in Python
  → Possible interfacing with islpy for affine transformations

About our PIL: Ivie

Main idea → Manipulate arrays in parallel programs

- Transpositions: data transposition, index permutation
- NUMA placements: interleaved allocation, replications

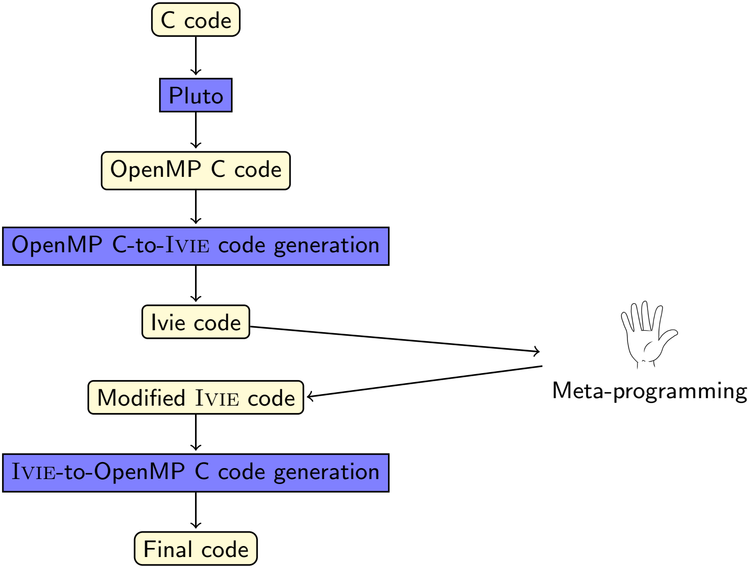
Design

- Declarative/functional
- Decoupled manipulation of array characteristics
- Physical and virtual memory abstraction
- Meta-language embedded in Python
  → Possible interfacing with islpy for affine transformations

What Ivie is not

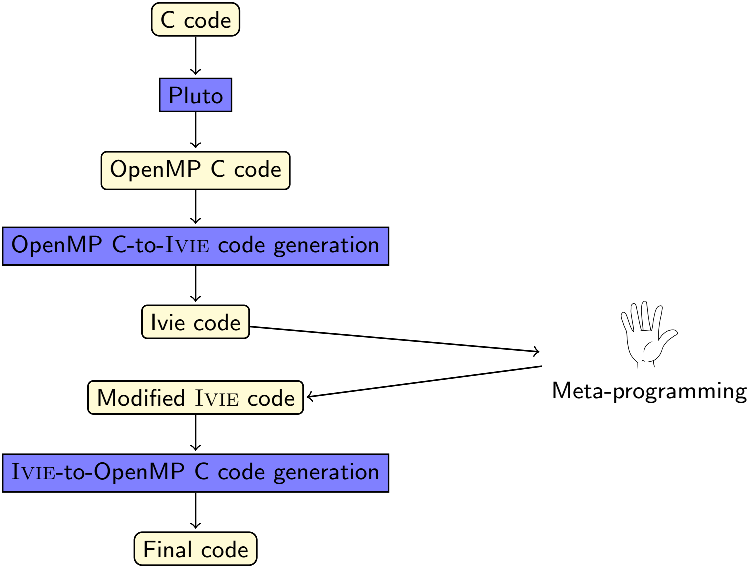
- A new programming language/domain-specific language

Implementation



Implementation

No more control flow optimization after Pluto!



Loop abstraction

Declaration of iterator types: parallel or sequential

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

Output variable

Arguments: array elements

- Implicit loop bounds
- Anonymous functions performing element-wise operations
- Accumulations made explicit
- Arrays follow either physical or virtual memory abstraction

Array declarations
Using default declaration construct

Declaration of array A

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

Parameters

- Number of dimensions
- Type
- Dimension sizes

- Used when generating code from input source

Input C code

```
int A[N][N];
int B[N][N];
int C[N][N];

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    C[i][j] = A[i][j] + B[i][j];
```

Output

```
A = array(2, int, [N,N])
B = array(2, int, [N,N])
C = array(2, int, [N,N])

with i as siter:
  with j as siter:
    C[i][j] = f(A[i][j], B[i][j])
```

Array declarations
Via data replication

Replication of array A

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

- Replication of read-only arrays
- `Ar` inherits all characteristics of `A`
  - Shape
  - Content
- Used when meta-programming

Replicating A and B

```
A = array(2, int, [N,N])
B = array(2, int, [N,N])
C = array(2, int, [N,N])

Ar = replicate(A)
Br = replicate(B)
```

Resulting C code

```
int A[N][N], B[N][N], C[N][N];
int Ar[N][N];
int Br[N][N];

for (i = 0; i < N; i++) {
  memcpy(Ar[i], A[i], N * sizeof(int));
  memcpy(Br[i], B[i], N * sizeof(int));
}
```

Array declarations
Via explicit transposition

Transposition of array A

```
A   = array(2, double, [N,N])
Atp = transpose(A, 1, 2)
```

Parameters

- Array of origin
- Dimension ranks to be permuted

Transposing A

```
A   = array(2, int, [N,N])
Atp = transpose(A, 1, 2)

with i as siter:
  with j as siter:
    ... = f(Atp[i][j], ...)
```

Resulting C code

```
int A[N][N];
int Atp[N][N];
/* Initialization of A */

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    Atp[i][j] = A[j][i];

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
     ... = Atp[i][j];
```

- `Atp` inherits from `A`
  - Content
  - Transposed shape
- `Atp` is **physical**
- Used when meta-programming

Array declarations

Via virtual index permutation

Transposition of array A

```
A   = array(2, double, [N,N])
Atv = vtranspose(A, 1, 2)
```

Parameters

- Array of origin
- Dimension ranks to be permuted

- `Atv` inherits from `A`
  - Content
  - Transposed shape
- `Atv` is **virtual**
- Used when meta-programming

Transposing A

```
A = array(2, int, [N,N])
Atv = vtranspose(A, 1, 2)

with i as siter:
  with j as siter:
    ... = f(Atv[i][j], ...)
```

Resulting C code

```
int A[N][N];

/* Initialization of A */

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
     ... = A[j][i];
```

Array declarations

For concise abstraction of several arrays

Abstracting arrays A and Ar

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

As = select([({it} <= val), A],
            [({it} > val), Ar])
```

Parameters

- Pairs of condition and arrays
- As is virtual
- Allows explicit control in partitioning
- For NUMA management

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

As = select([({it} <= val), A],
      [({it} > val), Ar])

with i as piter:
  with j as siter:
    ... = f(As[i][j])
```

```
int A[N][N];
int Ar[N][N];
/* Initialization of A and Ar */

#pragma omp parallel for schedule(...)
for (i = 0; i < N; i++)
  if (i <= val)
    for (j = 0; j < N; j++)
      ... = A[i][j];
  if (i > val)
    for (j = 0; j < N; j++)
      ... = Ar[i][j];
```

Data placement on NUMA

- Constructs based on API functions available in libnuma

Interleaved allocation

```
A = numa_alloc_interleaved(size)
```

```
A.map_interleaved(1)
```

Allocation on node

```
A = numa_alloc_onnode(size, node_id)
```

```
A.map_onnode(node_id)
```

Experimental setup

Machine	Intel Xeon E5-2697 v4 (Broadwell), 4 nodes, 36 cores
Compilation	`gcc -O3 -march=native` (enables vectorization)
Thread binding	`OMP_PROC_BIND`
Default Pluto options	Tiling for L1 cache, parallelization, vectorization

Experimental setup

Machine	Intel Xeon E5-2697 v4 (Broadwell), 4 nodes, 36 cores
Compilation	`gcc -O3 -march=native` (enables vectorization)
Thread binding	`OMP_PROC_BIND`
Default Pluto options	Tiling for L1 cache, parallelization, vectorization

Possible loop fusion heuristics:

- No loop fusion (no fuse)
- Maximum fusion (max fuse)
- In-between fusion (smart fuse)

Experimental setup

Machine	Intel Xeon E5-2697 v4 (Broadwell), 4 nodes, 36 cores
Compilation	`gcc -O3 -march=native` (enables vectorization)
Thread binding	`OMP_PROC_BIND`
Default Pluto options	Tiling for L1 cache, parallelization, vectorization

Possible loop fusion heuristics:

- No loop fusion (no fuse)
- Maximum fusion (max fuse)
- In-between fusion (smart fuse)

Different program versions:

- Default Pluto output (default)
- Pluto output + NUMA only (NUMA)
- Pluto output + transposition only (Layout)
- Pluto output + NUMA + transposition (NUMA-Layout)

Gemver

Code snippet

```
for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    x[i] = x[i] + beta * A[j][i] * y[j];
/* ... */
for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_N; j++)
    w[i] = w[i] +  alpha * A[i][j] * x[j];
```

Interesting properties

- Permutation profitable with loop fusions
- Several choices: need to find best permutation
- May loose some parallelism depending on chosen loop fusion
- Bandwidth-bound

Gemver

Meta-programs example: smart fuse vs no fuse

```
A = array(2, DATA_TYPE, [n, n])
u1 = array(1, DATA_TYPE, [n])
v1 = array(1, DATA_TYPE, [n])
A_v = vtranspose(A, 1, 2)
u1_1 = replicate(u1)
u1_2 = replicate(u1)
u1_3 = replicate(u1)

A.map_interleaved(1)
u1.map_onnode(0)
u1_1.map_onnode(1)
u1_2.map_onnode(2)
u1_3.map_onnode(3)

u1_s = select([0 <= {it} <= 8, u1],
              [9 <= {it} <= 17, u1_1],
              /*...*/)

with i as siter:
  with j as siter:
    A_v[i][j] = init()
```

No fuse

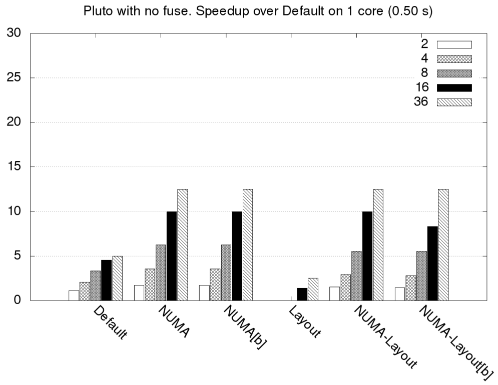
```
with t2 as cpiter:
  with t3 as siter:
    with t4 as siter:
      with t5 as siter:
        A[t4][t5] = f3(A[t4][t5], u1_s[t4],
                       v1[t5], u2_s[t4],
                       v2[t5])

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])
```

Smart fuse

```
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] = f7(x[t5], A[t4][t5], y[t4])
```

Gemver

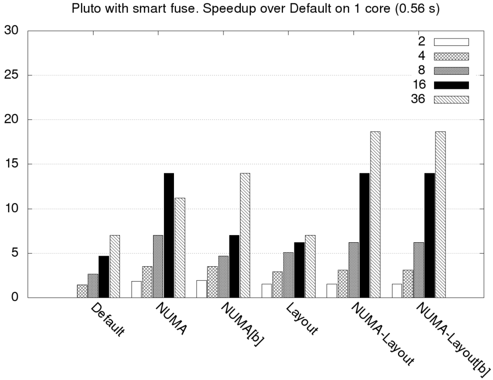
Different Pluto versions with no loop fusion



- ✓ More speed-up with NUMA
- × Much less speed-up with transposition
- × No added value with thread binding

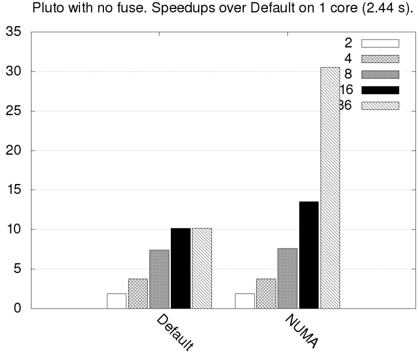
Gemver

Different Pluto versions with smart loop fusion



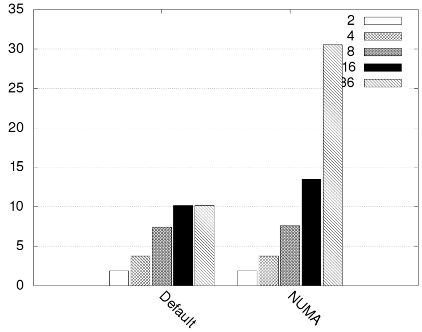
- ✓ More speed-up with NUMA
- ✓ More speed-up with transposition
- × No added value with thread binding

Gesummv

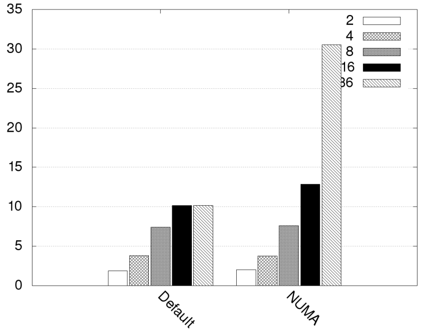

Pluto with no fuse. Speedups over Default on 1 core (2.44 s).
35
30
25
20
15
10
5
0
2
4
8
16
36
Default
NUMA

Gesummv

Pluto with no fuse. Speedups over Default on 1 core (2.44 s).



Pluto with max fuse. Speedups over Default on 1 core (2.44 s).



Gemm

Different naive versions

Interesting property

- Column-major access to B

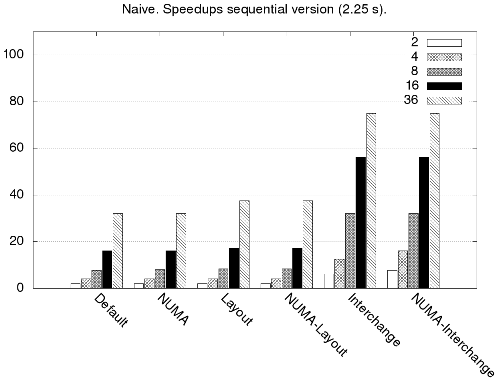
Modifications

- Transposed initialization of B
- NUMA placement: interleaved allocation only

```
# 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:
        with t7 as siter:
          with t6 as siter:
            C[t5][t6] = f9(C[t5][t6],
                        A[t5][t7], B[t6][t7])
```

Gemm

Different naive versions



- ✓ Some speed-up with transposition but loop interchange is better
- × No speed-up with NUMA

Gemm

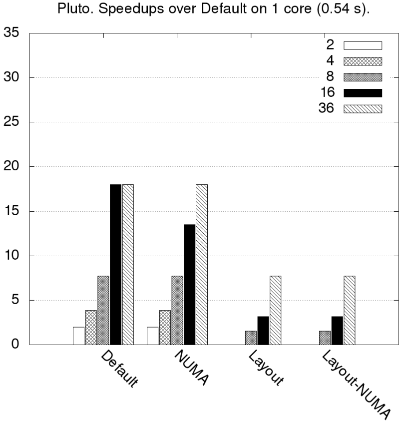
Different Pluto versions

Pluto's solution: loop interchange

Gemm

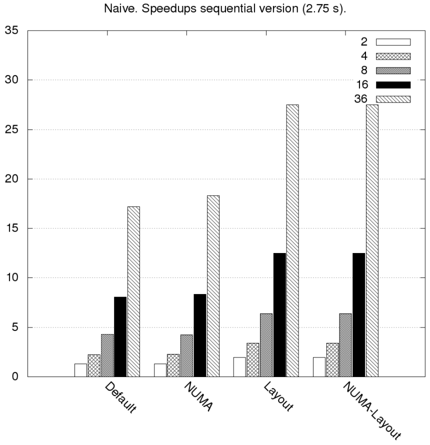
Different Pluto versions

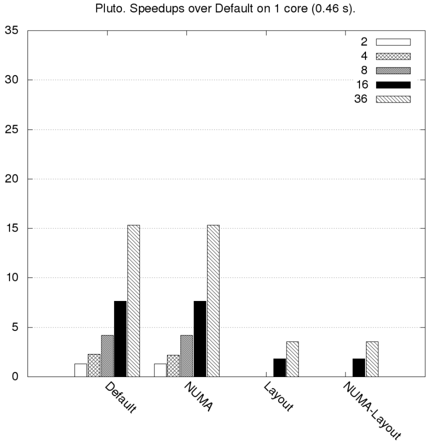
Pluto's solution: loop interchange



- × No speed-up with NUMA
- × Transposition makes things worse

Covariance


Naive. Speedups sequential version (2.75 s).
35
30
25
20
15
10
5
0
2
4
8
16
36
Default
NUMA
Layout
NUMA-Layout


Pluto. Speedups over Default on 1 core (0.46 s).
35
30
25
20
15
10
5
0
2
4
8
16
36
Default
NUMA
Layout
NUMA-Layout

Balance sheet

Advantages

- NUMA placements help bandwidth-bound programs
- More speed-up with transpositions
- New opportunities with transpositions

  $\rightarrow$ Wider space exploration for combining different types of optimizations

Balance sheet

Advantages

- NUMA placements help bandwidth-bound programs
- More speed-up with transpositions
- New opportunities with transpositions

  $\rightarrow$ Wider space exploration for combining different types of optimizations

Disadvantages

- Multiple conditional branching
- Copy overheads

Balance sheet

Advantages

- NUMA placements help bandwidth-bound programs
- More speed-up with transpositions
- New opportunities with transpositions
  $\rightarrow$ Wider space exploration for combining different types of optimizations

Disadvantages

- Multiple conditional branching
- Copy overheads

~~No more control flow optimization after Pluto!~~ Ok, we definitely still need some.

Some future work

Deeper investigation

- For case studies
- More experiments (other SCoPS, non-SCoPs)

Some future work

Deeper investigation

- For case studies
- More experiments (other SCoPS, non-SCoPs)

PIL design and implementation

- Revisit or extend some constructs
- Interfacing with islpy
- Memory and control flow optimizations: more integrated composition

Some future work

Deeper investigation

- For case studies
- More experiments (other SCoPS, non-SCoPs)

PIL design and implementation

- Revisit or extend some constructs
- Interfacing with islpy
- Memory and control flow optimizations: more integrated composition

Polyhedral analysis to help:

- determine interleaving granularity
- generate different schedules for transpositions

Some future work

Deeper investigation

- For case studies
- More experiments (other SCoPS, non-SCoPs)

PIL design and implementation

- Revisit or extend some constructs
- Interfacing with islpy
- Memory and control flow optimizations: more integrated composition

Polyhedral analysis to help:

- determine interleaving granularity
- generate different schedules for transpositions

A parallel intermediate language in Pluto's framework?

- Pure post-processing is difficult: Pluto outputs may be (very) complex.

Some future work

Deeper investigation

- For case studies
- More experiments (other SCoPS, non-SCoPs)

PIL design and implementation

- Revisit or extend some constructs
- Interfacing with islpy
- Memory and control flow optimizations: more integrated composition

Polyhedral analysis to help:

- determine interleaving granularity
- generate different schedules for transpositions

A parallel intermediate language in Pluto's framework?

- Pure post-processing is difficult: Pluto outputs may be (very) complex.
- **Ad hoc implementation probably the best solution for Pluto.**
  $\rightarrow$ But intermediate language necessary for space exploration of optimizations