AlphaZ: A System for Analysis, Transformation, and Code Generation in the Polyhedral Equational Model

Yuki et. Al. 2012, 2013

Malachi Phillips

Motivation for Polyhedral Model

- Some code optimizations are easy
    - Constant folding
    - Scalar replacement
- Others much harder
    - Loop inversion
    - Skewing
    - Tiling
- Production compilers may not do the latter
- Polyhedral model offers ways of reasoning about harder optimizations

Polyhedral Model

- Representation of subset of C known as ***static control programs***
  - **Do** statements with affine bounds
  - **if** conditionals with affine conditions
  - Bounds and conditionals depend on outer loop counters, constants

- ***Iteration Domain*** – set of values of the iteration vector for which to execute a statement
- ***Scattering function –*** an affine function specifying for each point in the iteration domain, a new coordinate for a corresponding statement instance

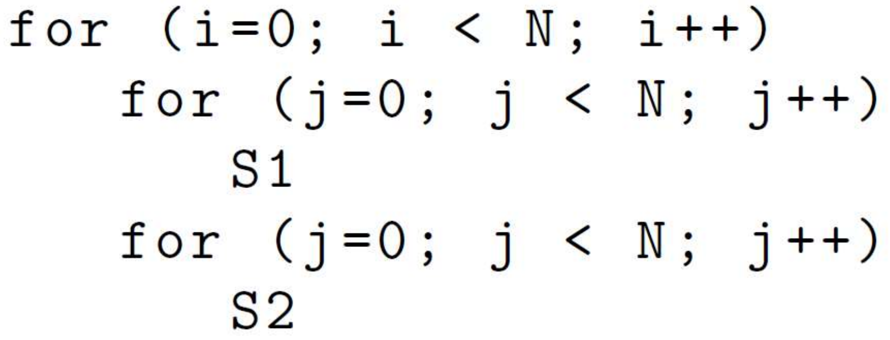
Scattering Function

- Several interpretations:
  - Distribute iterations in space, i.e. processors, order in time, or both
  - ***Space mapping*** – number corresponds to processor for executing statement
  - ***Time mapping*** – order statements in some lexicographical order

- ***Target Mapping*** – execution strategies
  - Schedule
  - Memory Mapping
  - Processor allocation
  - Tiling
  - Expressed as dependencies between indices

Example Target Mappings

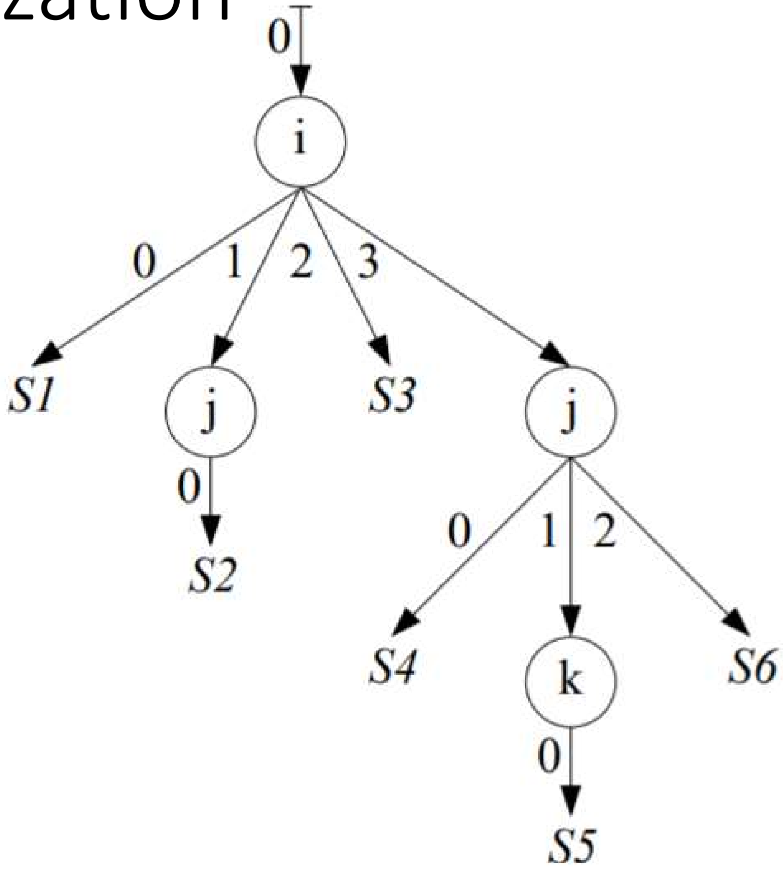
- $\theta = (i, j \to i, j)$
  - Identity mapping
- $\theta = (i, j, k \to i, j)$
  - Projection onto i, j plane
- $\theta_{S1} = (i, j \to i, 0, j)$, $\theta_{S2} = (i, j \to i, 1, j)$



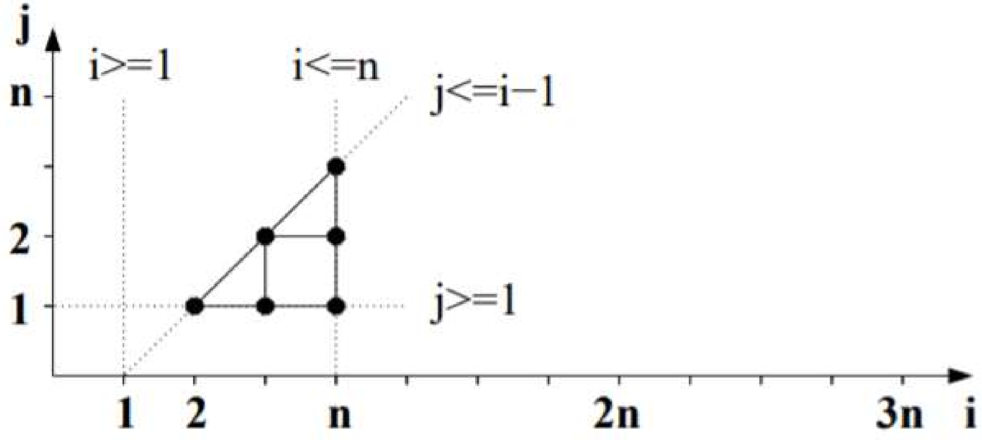
```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        S1
    for (j=0; j < N; j++)
        S2
```

Example: Cholesky Factorization

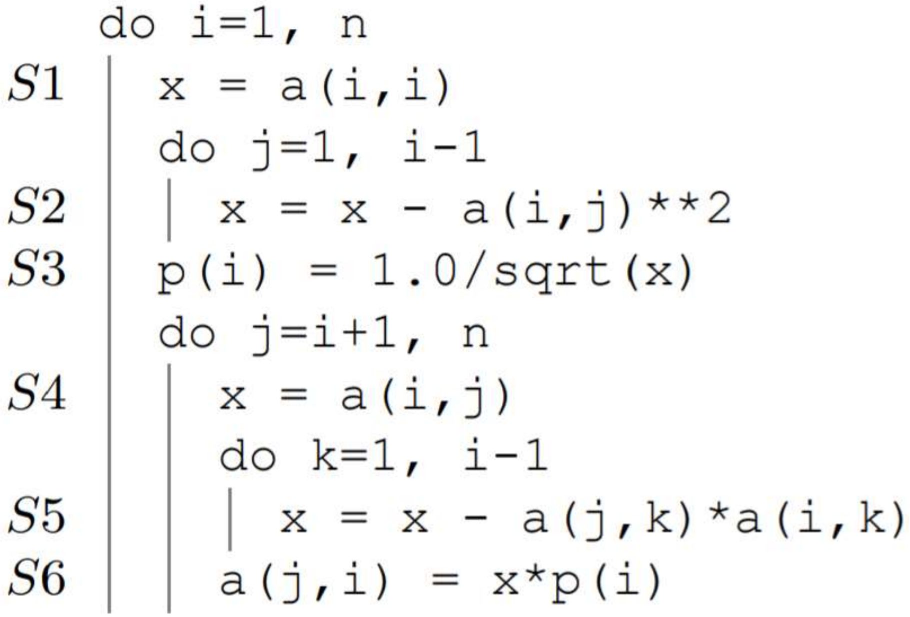
```
    do i=1, n
S1  |  x = a(i,i)
    |  do j=1, i-1
S2  |  |  x = x - a(i,j)**2
S3  |  p(i) = 1.0/sqrt(x)
    |  do j=i+1, n
S4  |  |  x = a(i,j)
    |  |  do k=1, i-1
S5  |  |  |  x = x - a(j,k)*a(i,k)
S6  |  |  a(j,i) = x*p(i)
```



[Bastoul 2004]



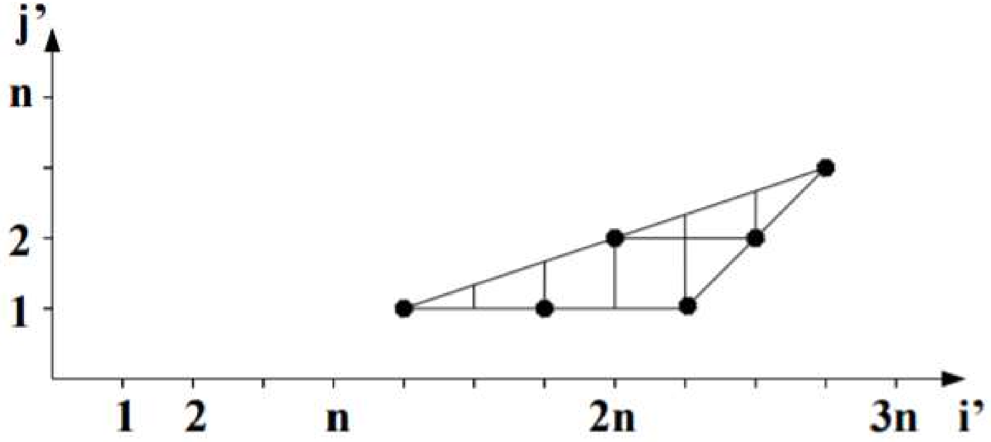
S2 bounds



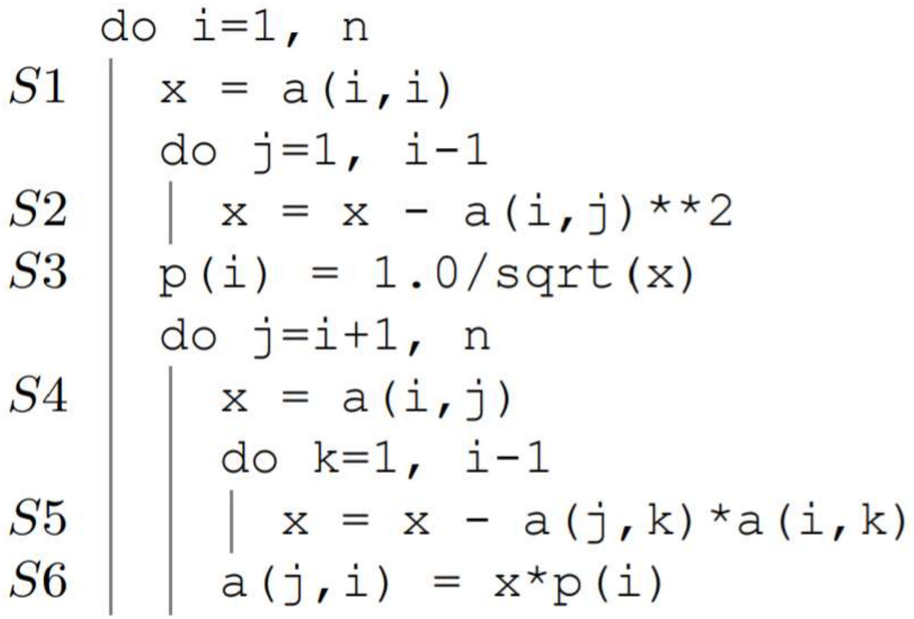
```
    do i=1, n
S1  | x = a(i,i)
    | do j=1, i-1
S2  | | x = x - a(i,j)**2
S3  | p(i) = 1.0/sqrt(x)
    | do j=i+1, n
S4  | | x = a(i,j)
    | | do k=1, i-1
S5  | | | x = x - a(j,k)*a(i,k)
S6  | | a(j,i) = x*p(i)
```

[Bastoul 2004]

$$\theta_{S2} = (i, j \rightarrow 2i, j)$$



Modified S2 bounds



```
   do i=1, n
S1   x = a(i,i)
     do j=1, i-1
S2     x = x - a(i,j)**2
S3   p(i) = 1.0/sqrt(x)
     do j=i+1, n
S4     x = a(i,j)
       do k=1, i-1
S5       x = x - a(j,k)*a(i,k)
S6     a(j,i) = x*p(i)
```

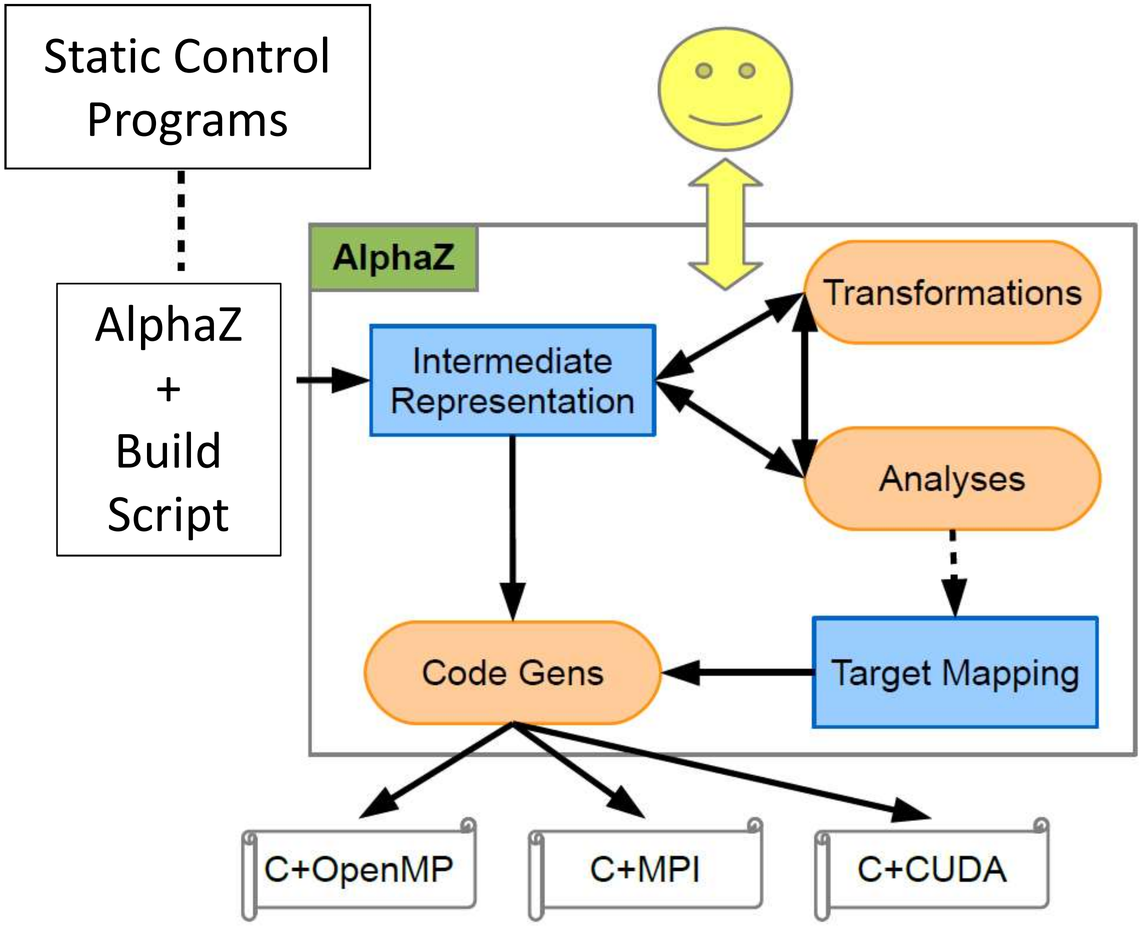
[Bastoul 2004]

Pros/Cons of Polyhedral Compilation

- Pros:
  - Detailed representation, analysis, optimization
  - Loop transformation
- Cons:
  - Polyhedral “search-space” is quite large
  - AlphaZ – leave all of this for the user!

Why another polyhedral code transformation tool?

- Supports Parametric tiling
  - Tile sizes not fixed at compile time
  - Enables empirical search for tile sizes
- Automatically manages re-allocation of memory
  - “None of the existing polyhedral parallelizers for distributed memory even mention data partitioning. Instead, they use the same memory allocation as the original sequential program on all nodes.” [Yuki 2013]
- Focus on distributed memory parallelism
- Polyhedral machinery to:
  - Apply loop transformations to expose coarse grained parallelism
  - **User defined target mapping – error prone!**
    - **e.g.** Set in build script: setMemoryMap(program, system, “var1”, “var2”, (i,j,k->i,j))



Static Control Programs

AlphaZ + Build Script

AlphaZ

Intermediate Representation

Transformations

Analyses

Code Gens

Target Mapping

C+OpenMP

C+MPI

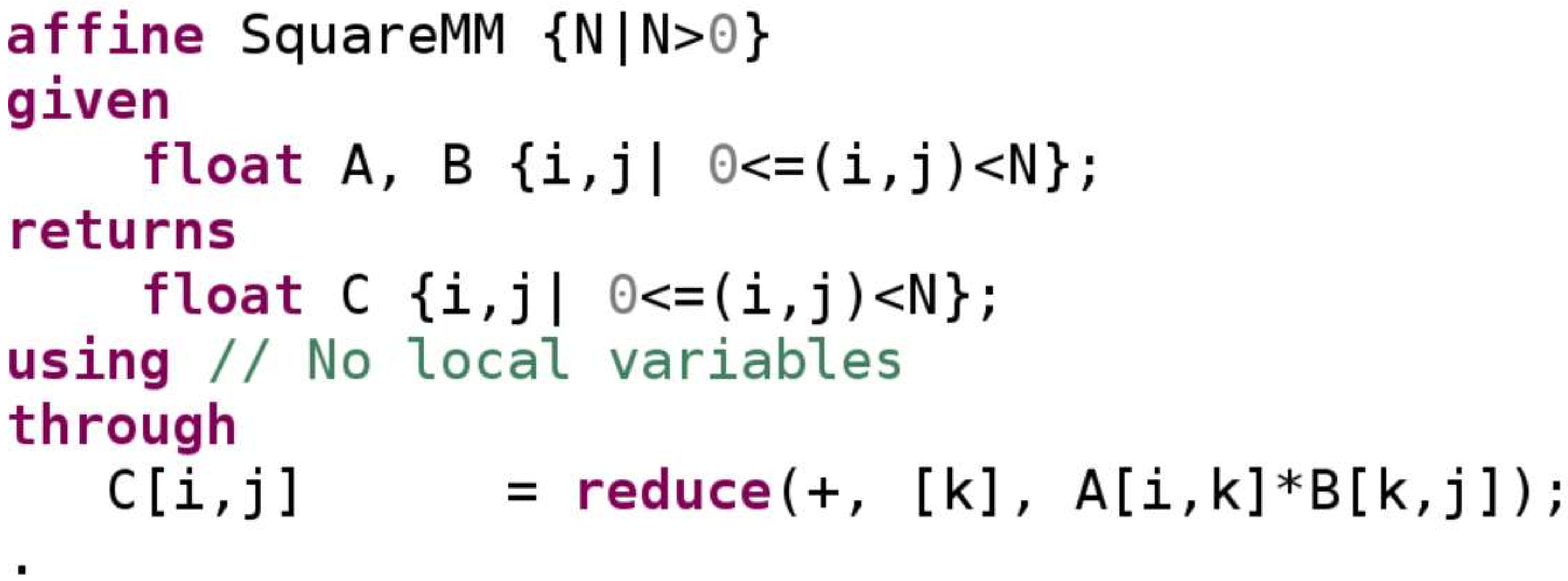
C+CUDA

[Yuki 2013]

Human-in-the-loop

- Automatic parallelization is ultimate goal
    - Yuki 2013 claims automatic tools are “restrictive”
        - Difficult to surpass hard coded, domain specific knowledge
- AlphaZ aims to provide full control to the user
    - Generate new transformations quickly, mostly through provided build script
    - Guide transformations with domain specific knowledge
    - Any performance benchmark using AlphaZ required a human to find those combinations of polyhedral transformations

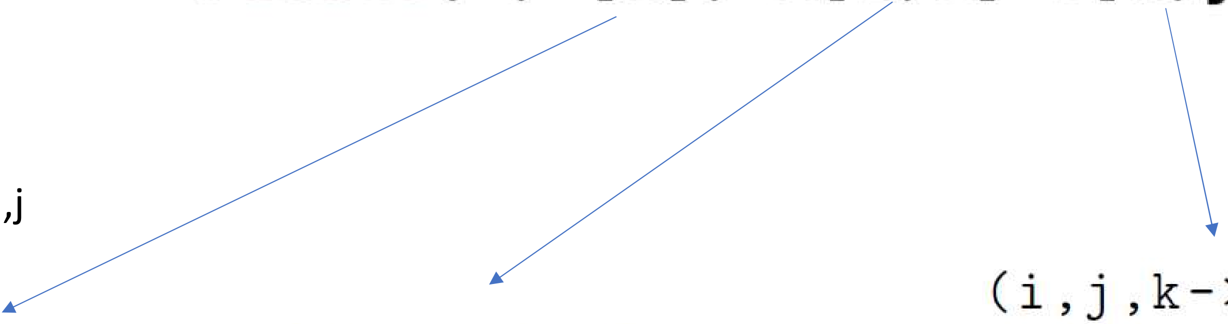
“Hello, world!” Matrix Multiply in AlphaZ



```
affine SquareMM {N|N>0}
given
    float A, B {i,j| 0<=(i,j)<N};
returns
    float C {i,j| 0<=(i,j)<N};
using // No local variables
through
   C[i,j]      = reduce(+, [k], A[i,k]*B[k,j]);
.
```

- AlphaZ source code (what the user writes)
- N becomes runtime parameter in generated C code

```
C[i,j]      = reduce(+, [k], A[i,k]*B[k,j]);
```



```
(i,j,k->i,j)
```

Project indices i,j,k onto i,j (projection function)

Maps $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$

```
(i,j,k->i,k)@A
```

In the entire iteration space with i,j,k, only give me i,k (projection on i-k plane)

```
(i,j,k->k,j)@B)
```

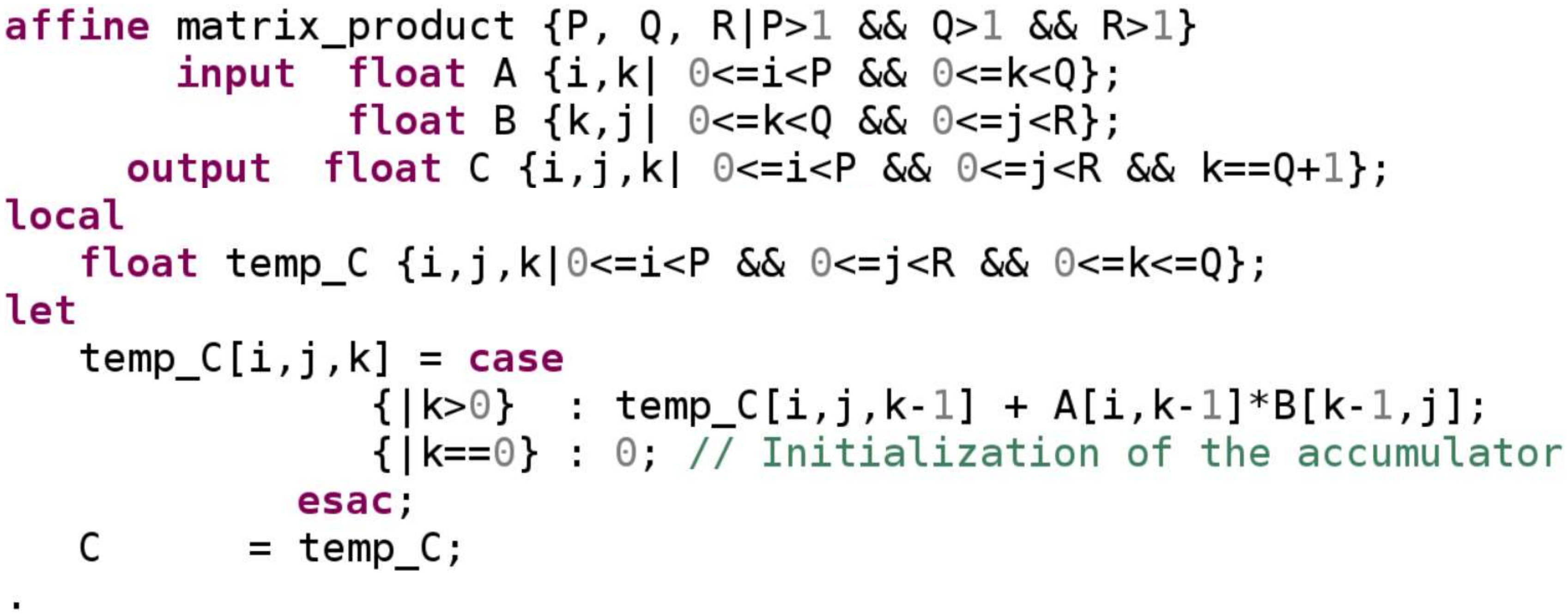
In the entire iteration space with i,j,k, only give me k,j (projection onto k-j plane)

When multiple points in the left-hand side are mapped to a same point in the right-hand side, those values are combined using the reduction operator.

Equivalent to:

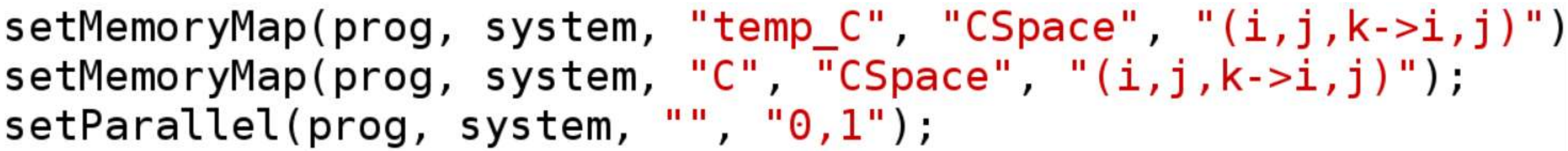
```
C = reduce(+, (i,j,k->i,j), (i,j,k->i,k)@A * (i,j,k->k,j)@B);
```

Parallel Matrix-Matrix Multiply



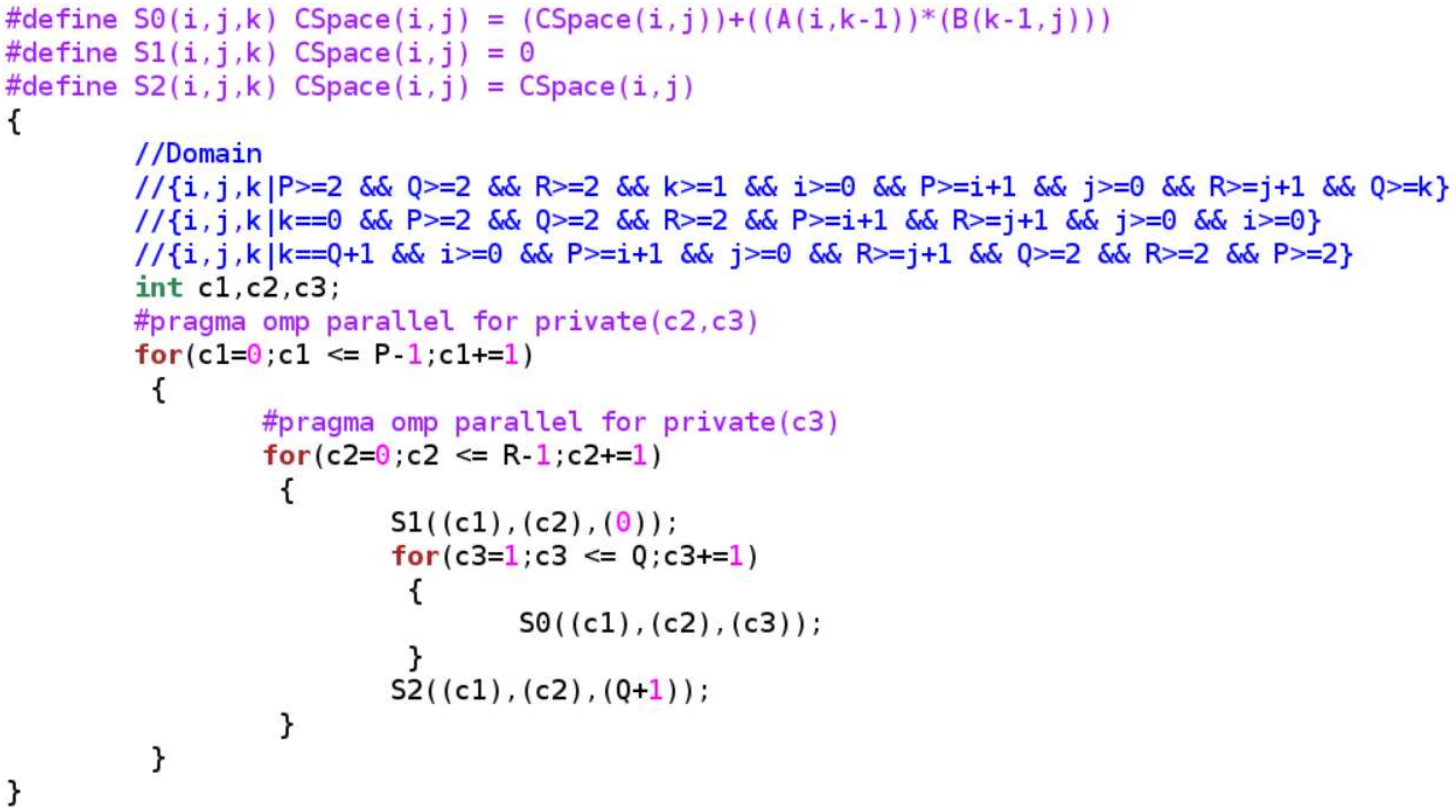
```
affine matrix_product {P, Q, R|P>1 && Q>1 && R>1}
       input  float A {i,k| 0<=i<P && 0<=k<Q};
              float B {k,j| 0<=k<Q && 0<=j<R};
     output  float C {i,j,k| 0<=i<P && 0<=j<R && k==Q+1};
local
   float temp_C {i,j,k|0<=i<P && 0<=j<R && 0<=k<=Q};
let
   temp_C[i,j,k] = case
               {|k>0}  : temp_C[i,j,k-1] + A[i,k-1]*B[k-1,j];
               {|k==0} : 0; // Initialization of the accumulator
            esac;
   C      = temp_C;
.
```

3rd dimension of array is a fictitious “time” dimension

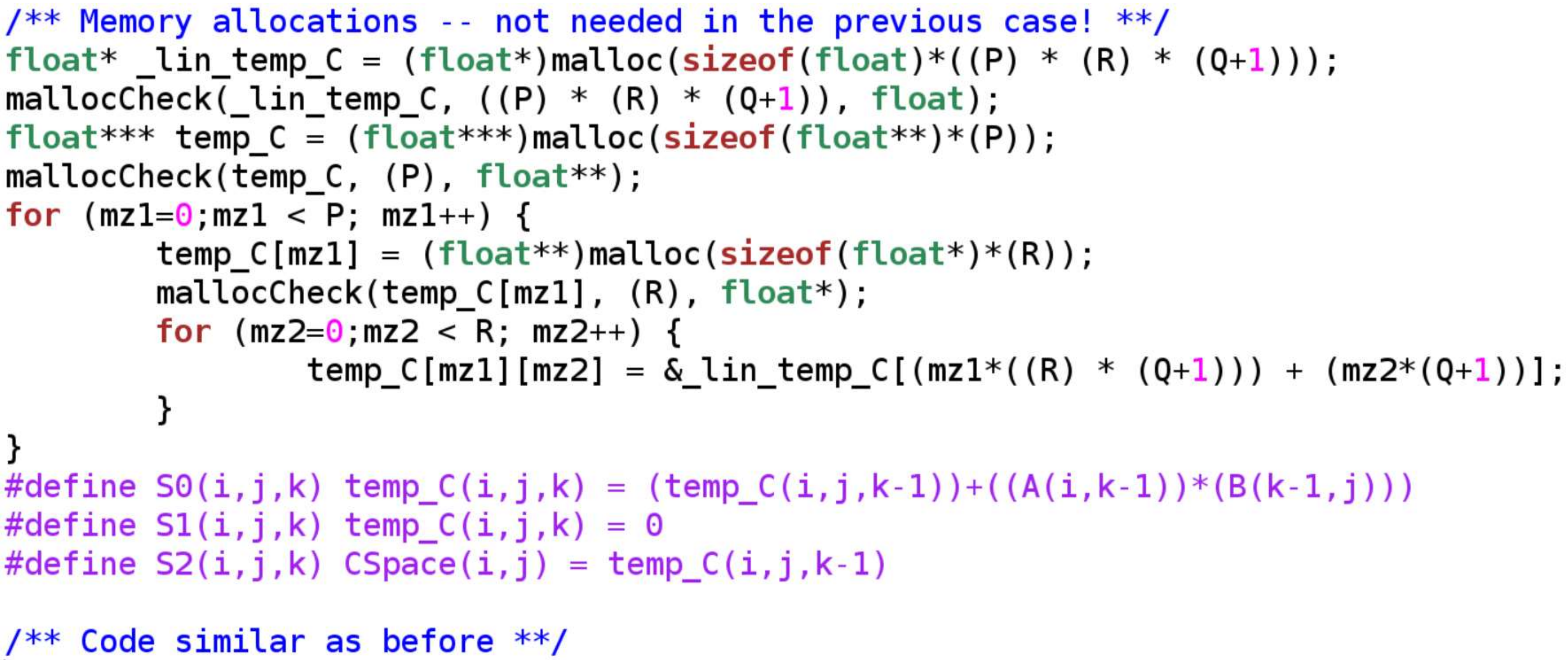


```
setMemoryMap(prog, system, "temp_C", "CSpace", "(i,j,k->i,j)")
setMemoryMap(prog, system, "C", "CSpace", "(i,j,k->i,j)");
setParallel(prog, system, "", "0,1");
```

- (Almost) minimum viable build script for this case
  - If one removes either of the memory maps for temp_C, C, generated c code *will* generate a 3D temporary array
  - This is where the user specifies mappings, changes statement orderings, processor allocation, tiling
- (i,j,k->i,j) repeated in both memory maps indicates a reduction
- CSpace represents actual matrix-matrix product result
- AlphaZ verifier can catch when dependency is broken



```
#define S0(i,j,k) CSpace(i,j) = (CSpace(i,j))+((A(i,k-1))*(B(k-1,j)))
#define S1(i,j,k) CSpace(i,j) = 0
#define S2(i,j,k) CSpace(i,j) = CSpace(i,j)
{
        //Domain
        //{i,j,k|P>=2 && Q>=2 && R>=2 && k>=1 && i>=0 && P>=i+1 && j>=0 && R>=j+1 && Q>=k}
        //{i,j,k|k==0 && P>=2 && Q>=2 && R>=2 && P>=i+1 && R>=j+1 && j>=0 && i>=0}
        //{i,j,k|k==Q+1 && i>=0 && P>=i+1 && j>=0 && R>=j+1 && Q>=2 && R>=2 && P>=2}
        int c1,c2,c3;
        #pragma omp parallel for private(c2,c3)
        for(c1=0;c1 <= P-1;c1+=1)
         {
                #pragma omp parallel for private(c3)
                for(c2=0;c2 <= R-1;c2+=1)
                 {
                        S1((c1),(c2),(0));
                        for(c3=1;c3 <= Q;c3+=1)
                         {
                                S0((c1),(c2),(c3));
                         }
                        S2((c1),(c2),(Q+1));
                 }
         }
}
```



```
/** Memory allocations -- not needed in the previous case! **/
float* _lin_temp_C = (float*)malloc(sizeof(float)*((P) * (R) * (Q+1)));
mallocCheck(_lin_temp_C, ((P) * (R) * (Q+1)), float);
float*** temp_C = (float***)malloc(sizeof(float**)*(P));
mallocCheck(temp_C, (P), float**);
for (mz1=0;mz1 < P; mz1++) {
        temp_C[mz1] = (float**)malloc(sizeof(float*)*(R));
        mallocCheck(temp_C[mz1], (R), float*);
        for (mz2=0;mz2 < R; mz2++) {
                temp_C[mz1][mz2] = &_lin_temp_C[(mz1*((R) * (Q+1))) + (mz2*(Q+1))];
        }
}
#define S0(i,j,k) temp_C(i,j,k) = (temp_C(i,j,k-1))+((A(i,k-1))*(B(k-1,j)))
#define S1(i,j,k) temp_C(i,j,k) = 0
#define S2(i,j,k) CSpace(i,j) = temp_C(i,j,k-1)

/** Code similar as before **/
```

Result of commenting out the top line of the build script

Performance Evaluation

- Polybench
- Cray XT6m, CrayCC/5.04 –O3
- PLuTo comparison: --tile –parallel –noprevector
- Manually explore different tile sizes in PLuTo and AlphaZ
- No comparison to hand-tuned implementation, absolute performance
- Performance normalized to speed up over single core PLuTO implementation

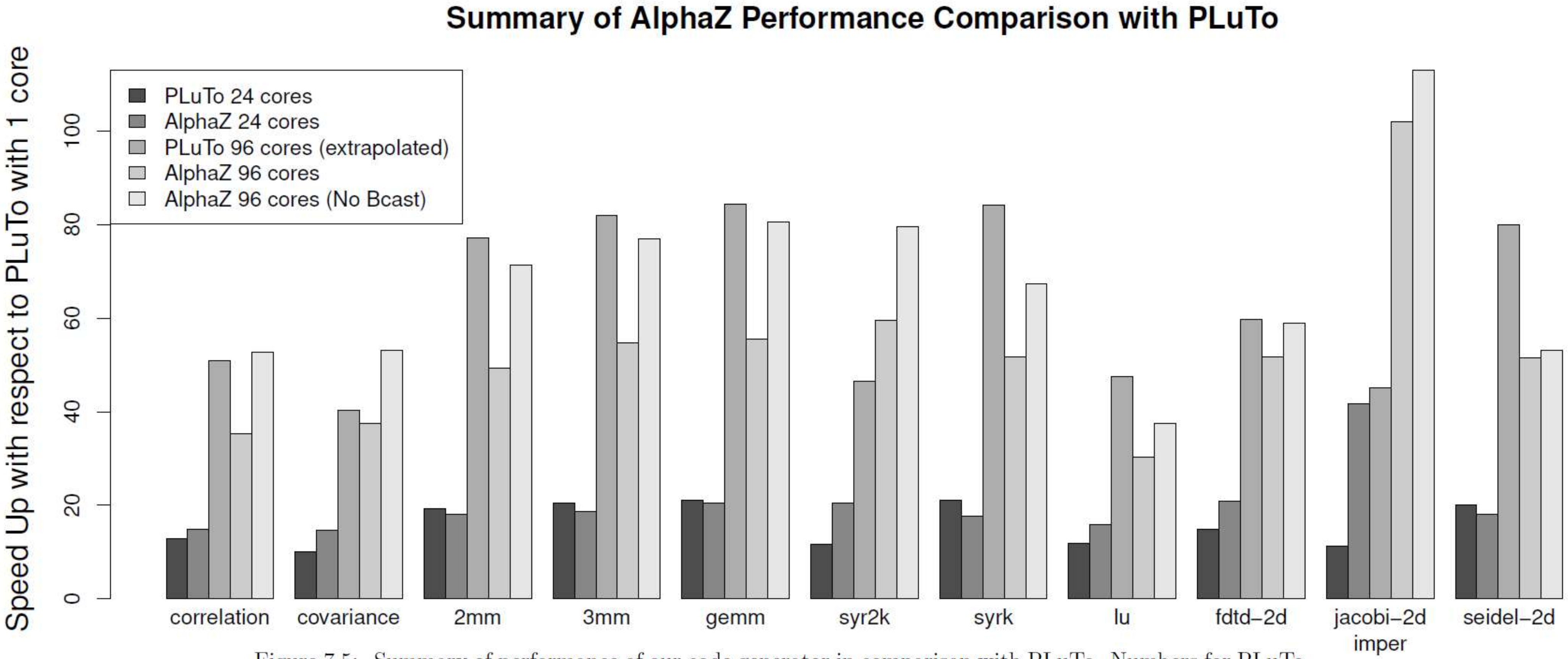
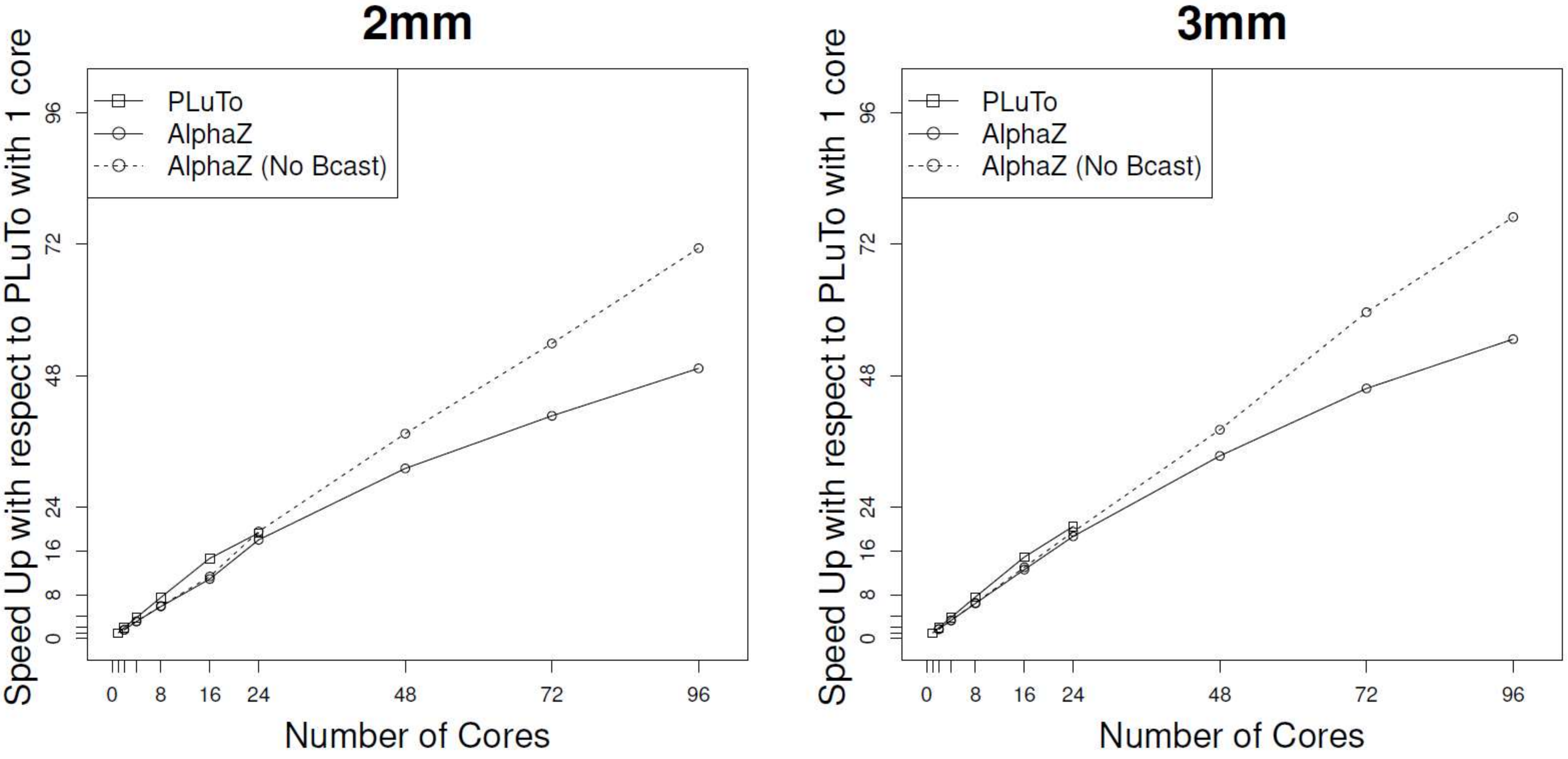
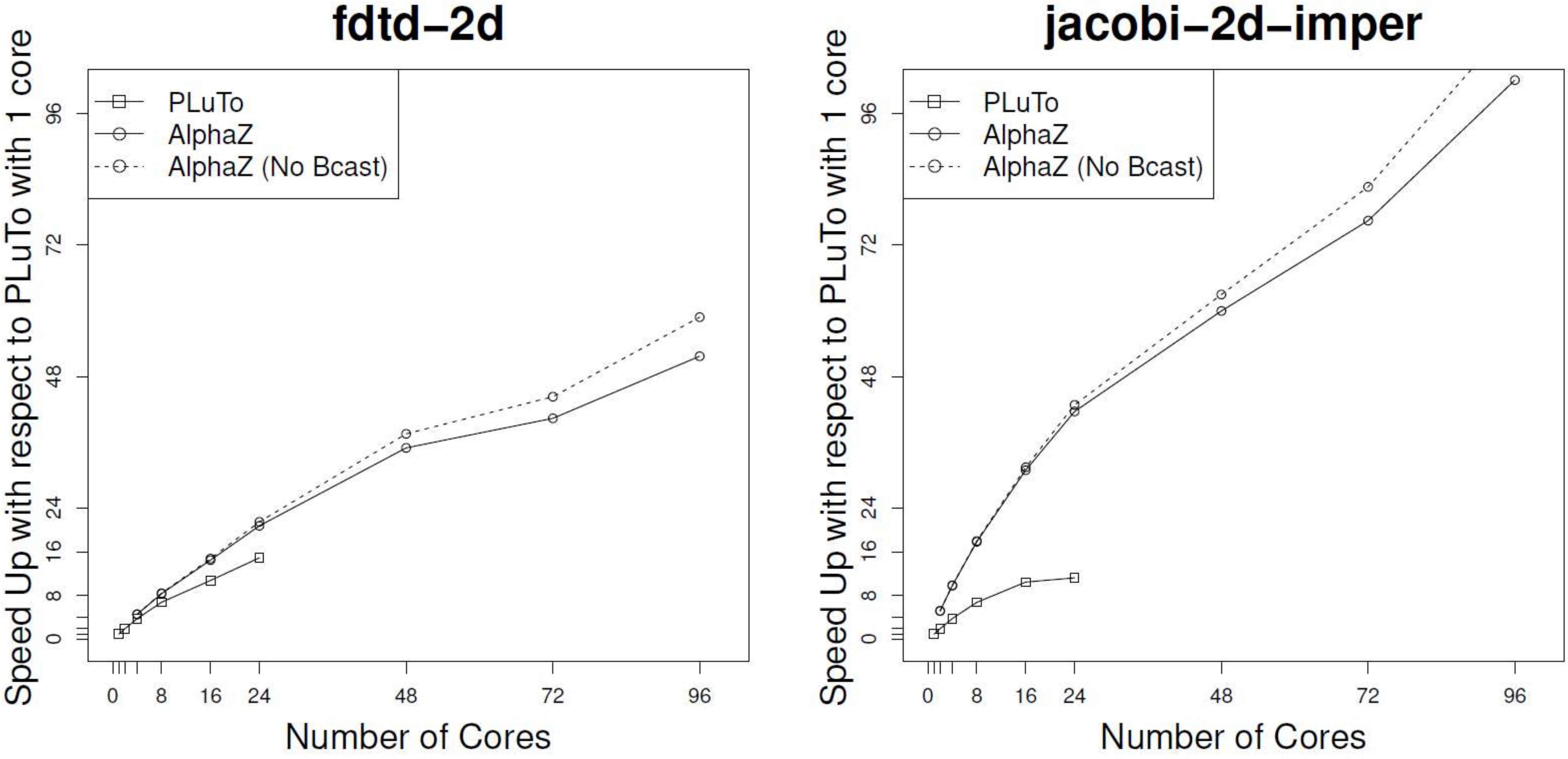


Figure 7.5: Summary of performance of our code generator in comparison with PLuTo. Numbers for PLuTo with 96 cores are extrapolated (multiplied by 4) from the speed up with 24 cores. MPIC (No Bcast) are number with the time to broadcast inputs removed. Broadcast of inputs takes no more than 2.5 seconds, but have strong impact on performance for the problem sizes used. This is a manifestation of the well known Amdahl's Law, and is irrelevant to weak scaling that we focus on. With the cost of broadcast removed, our code generator matches or exceeds the scaling of shared memory parallelization by PLuTo.

[Yuki 2013]



Speed up compared to single core PLuTo implementation for matrix-matrix multiply [Yuki 2013]
Left: two matrix-matrix multiplies. *Right:* three matrix-matrix multiplies



Speed up compared to single core PLuTo implementation for stencil computations [Yuki 2013]
Left: 2d finite difference time domain kernel. *Right:* 2-D jacobi stencil computation.

Conclusion

Pros:

- Expose target mapping to user to allow exploration of polyhedral code generation
- Can significantly outperform PLuTo for certain kinds of workloads (stencils)
- Represented through simple Alpha language + transformation script

Cons:

- User-specified target mappings may not result in correct implementation
- Only roughly as performant as PLuTo for some workloads (linear algebra)

Sources

T. Yuki, ‘Beyond Shared Memory Loop Parallelism in the Polyhedral Model’, Colorado State University, 2013.

C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, Antibes Juan-les-Pins, France, 2004, pp. 7–16.

T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, “AlphaZ: A System for Design Space Exploration in the Polyhedral Model,” in *Languages and Compilers for Parallel Computing*, vol. 7760, H. Kasahara and K. Kimura, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 17–31.