Scalable, Robust, and “Regression-Free” Loop Optimizations for Scientific Fortran and Modern C++

Michael Kruse, Tobias Grosser

Albert Cohen, Sven Verdoolaege, Oleksandre Zinenko
Polly Labs, ENS Paris

Johannes Doerfert
Uni. Saarbruecken

Siddharth Bhat
IIIT Hydrabad (Intern at ETH)

Roman Gereev,
Ural Federal University

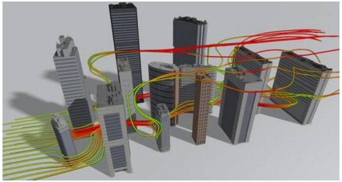
Hongin Zheng, Alexandre Isonard
Xilinx

Swiss Universities / PASC
Qualcomm, ARM, Xilinx

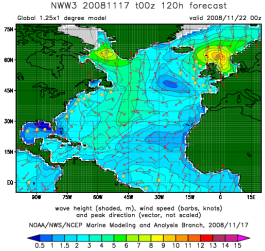
… many others

polly labs ARM ETH Zürich

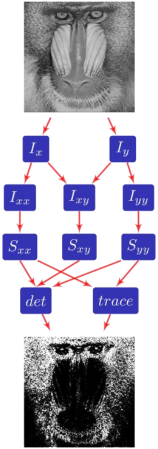
Physics Simulations



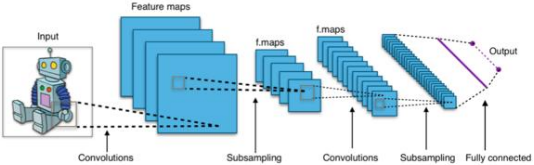
Weather



Graphics



Machine Learning



COSMO: Weather and Climate Model

- 500.000 Lines of Fortran
- 18.000 Loops
- 19 Years of Knowledge

- Used in Switzerland, Russia, Germany, Poland, Italy, Israel, Greece, Romania, …

COSMO – Climate Modeling

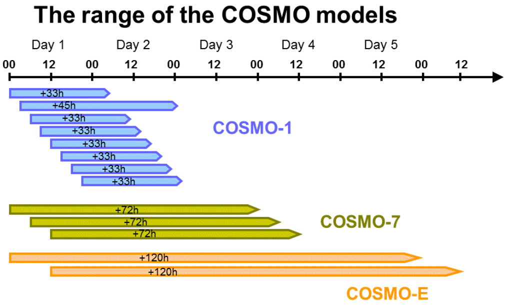
Piz Daint, Lugano, Switzerland

TOP 500 The List.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini	560,640	17,590.0	27,112.5	8,209

- Global (low-resolution Model)
- Up to 5000 Nodes
- Run ~monthly

COSMO – Weather Forecast



- Regional model
- High-resolution
- Runs hourly
  (20 instances in parallel)

- Today: 40 Nodes * 8 GPU
- Manual Translation to GPUs
- 3 year multi-person project

Can LLVM do
this automatically?

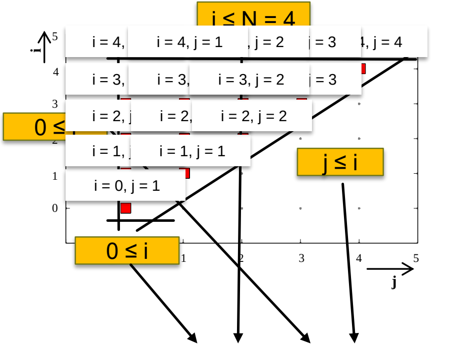
Polyhedral Model – In a nutshell

Program Code

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= i; j++)
    S(i,j);
```

N = 4

Iteration Space



D = { (i,j) | 0 ≤ i ≤ N ∧ 0 ≤ j ≤ i }

Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation
Tobias Grosser, Armin Groesslinger, and Christian Lengauer in Parallel Processing Letters (PPL), April, 2012

Statistics - COSMO

Siddharth Bhat

- **Number of Loops**
  - 18,093 Total
  - 9,760 Static Control Loops (Modeled precisely by Polly)
  - 15,245 Non-Affine Memory Accesses (Approximated by Polly)
  - 11.154 Loops after precise modeling, less e.g. due to:
    - *Infeasible assumptions taken, or modeling timeouts*
- **Largest set of loops:** 72 loops
- **Reasons why loops cannot be modeled**
  - Function calls with side-effects
  - Uncomputable loops bounds (data-dependent loop bounds?)

Interprocedural Loop Interchange for GPU Execution

Pulled out parallel loop for OpenACC Annotations

```
#ifdef _OPENACC
  !$acc parallel
  !$acc loop gang vector
  DO j1 = ki1sc, ki1ec
    CALL coe_th_gpu(pduh2oc (j1, ki3sc), pduh2of(j1, ki3sc), pduco2(j1, ki3sc),
                    pduo3(j1, ki3sc), …, pa2f(j1), pa3c(j1), pa3f(j1))
  ENDDO
  !$acc end parallel
#else
  CALL coe_th (pduh2oc, pduh2of, pduco2, pduo3, palogp, palogt, podsc, podsf, podac, podaf,
               …, pa3c, pa3f)
#endif
```

Optical Effect on Solar Layer

Outer loop is sequential

Inner loop is parallel

```
DO j3 = ki3sc+1, ki3ec
   CALL coe_th (j3) {    ! Determine effect of the layer in *coe_th*
   ! Optical depth of gases
     DO j1 = ki1sc, ki1ec
       …
       IF (kco2 /= 0) THEN
         zodgf = zodgf + pduco2(j1   ,j3)* (cobi(kco2,kspec,2)* EXP ( coali(kco2,kspec,2) *
               palogp(j1   ,j3) -cobti(kco2,kspec,2) * palogt(j1   ,j3)))
       ENDIF
     …
     zeps=SQRT(zodgf*zodgf)
    …
    ENDDO
   }

   DO j1 = ki1sc, ki1ec   ! Set RHS
         …
   ENDDO
   DO j1 = ki1sc, ki1ec   ! Elimination and storage of utility variables
         …
   ENDDO
 ENDDO     ! End of vertical loop over layers
```

Sequential Dependences

Inner loop is parallel

Inner loop is parallel

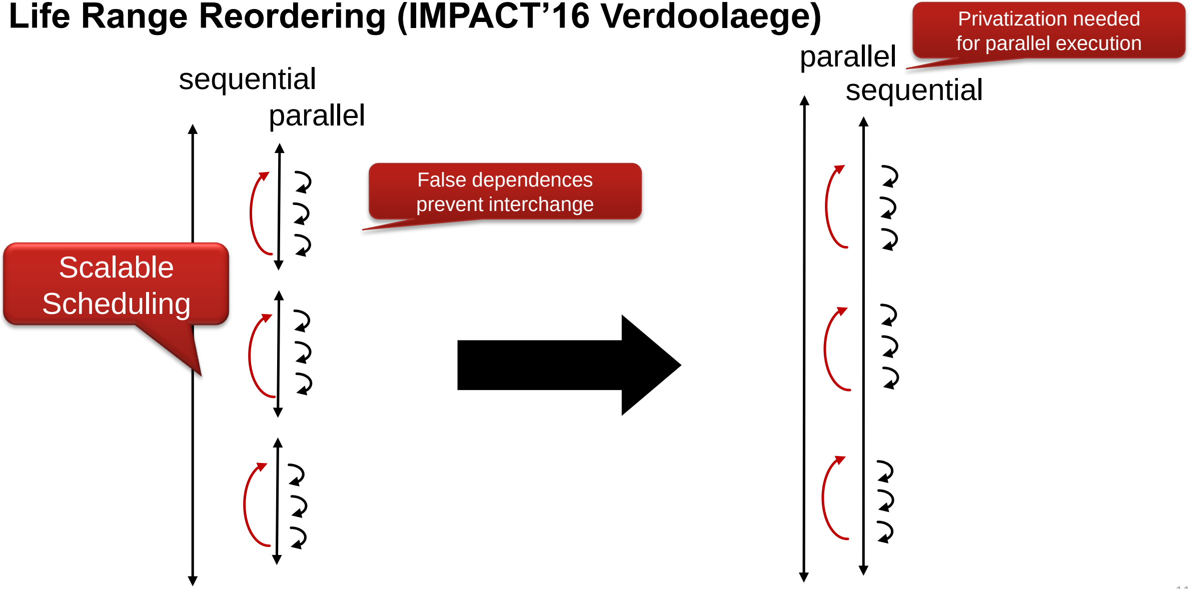
Optical Effect on Solar Layer – After interchange

```
!> Turn loop structure with multiple ip loops inside a
!> single k loop into perfectly nested k-in loop on GPU
#ifdef _OPENACC
  !$acc parallel
  !$acc loop gang vector
  DO j1 = ki1sc, ki1ec
    !$acc loop seq
    DO j3 = ki3sc+1, ki3ec         ! Loop over vertical

    ! Determine effects of layer in *coe_so*
      CALL coe_so_gpu(pduh2oc (j1,j3)   , pduh2of (j1,j3)   , …, pa4c(j1), pa4f(j1), pa5c(j1), pa5f(j1))

    ! Elimination
     …
      ztd1 = 1.0_dp/(1.0_dp-pa5f(j1)*(pca2(j1,j3)*ztu6(j1,j3-1)+pcc2(j1,j3)*ztu8(j1,j3-1)))
             ztu9(j1,j3) = pa5c(j1)*pcd1(j1,j3)+ztd6*ztu3(j1,j3) + ztd7*ztu5(j1,j3)
    ENDDO
END DO       ! Vertical loop
  !$acc end parallel
```

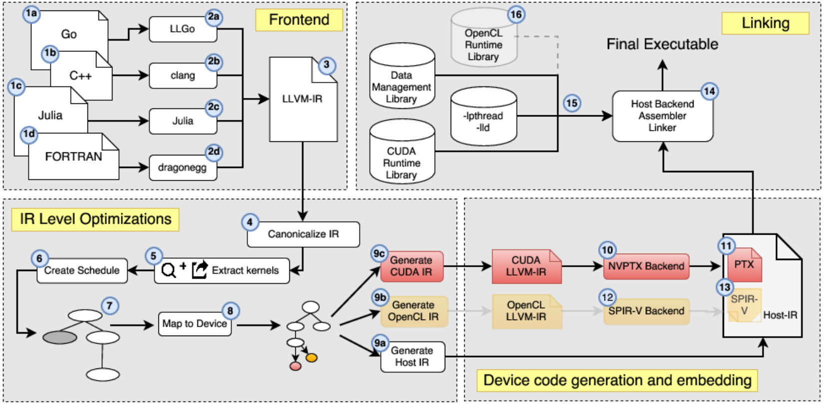
Outer loop is parallel

Inner loop is sequential

Life Range Reordering (IMPACT'16 Verdoolaege)



Polly-ACC: Architecture



Polly-ACC: Transparent Compilation to Heterogeneous Hardware
Tobias Grosser, Torsten Hoefler at International Conference of Supercomputing (ICS), June 2016, Istanbul

Polly-ACC: Architecture

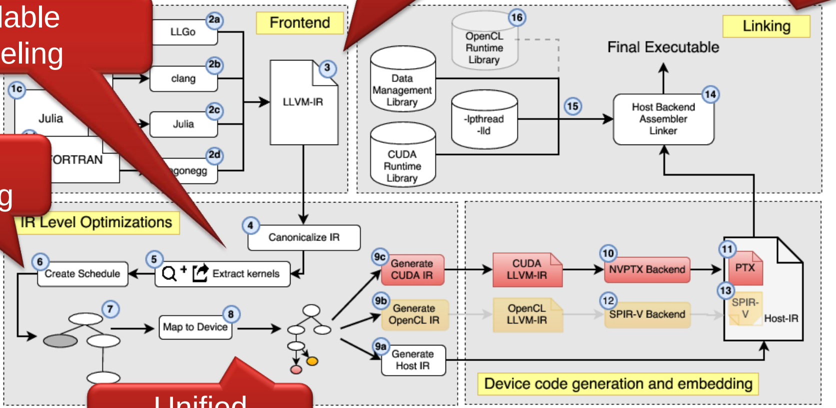
Intrinsics to model Multi-dimensional strided arrays

Better ways to link with NVIDIA libdevice

Scalable Modeling

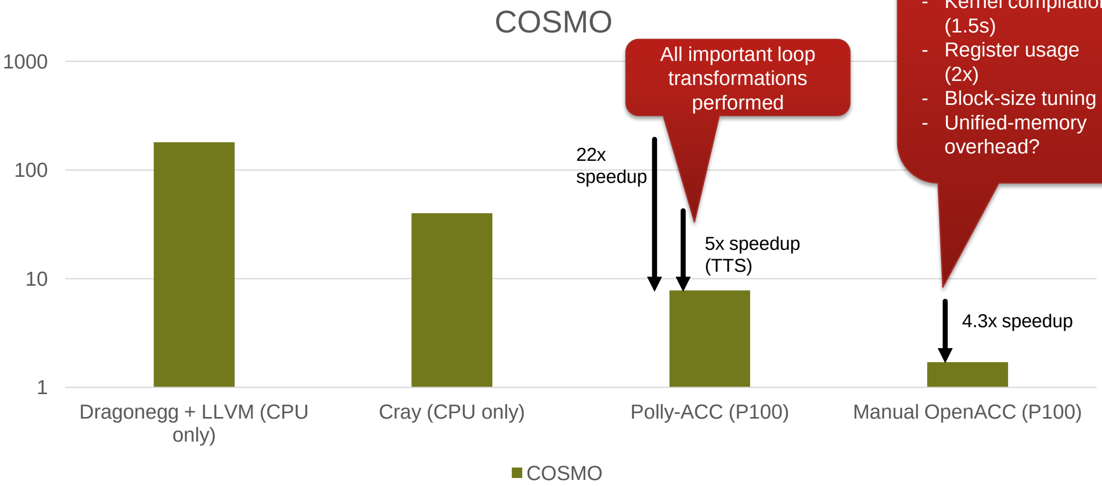
Scalable Scheduling

Unified Memory



Polly-ACC: Tr… …ous Hardware
Tobias Grosser, Torsten Hoefler at International Conference of Supercomputing (ICS), June 2016, Istanbul

Performance



Headroom:

- Kernel compilation (1.5s)
- Register usage (2x)
- Block-size tuning
- Unified-memory overhead?

Expression Templates (in a nutshell)

```
class Vec : public VecExpression<Vec> {
    std::vector<double> elems;

  public:
    double operator[](size_t i) const { return elems[i]; }
    double &operator[](size_t i)      { return elems[i]; }
    size_t size() const               { return elems.size(); }

    Vec(size_t n) : elems(n) {}

    // A Vec can be constructed from any VecExpression, forcing its evaluation.
    template <typename E>
    Vec(VecExpression<E> const& vec) : elems(vec.size()) {
        for (size_t i = 0; i != vec.size(); ++i) {
            elems[i] = vec[i];
        }
    }
};
```

Expression Templates (in a nutshell)

```
class Vec : public VecExpression<Vec> {
    std::vector<double> elems;

  public:
    double operator[](size_t i) const { return elems[i]; }
    double &operator[](size_t i)      { return elems[i]; }
    size_t size() const               { return elems.size(); }

    Vec(size_t n) : elems(n) {}

    // A Vec can be constructed from any VecExpression, forcing its evaluation.
    template <typename E>
    Vec(VecExpression<E> const& vec) : elems(vec.size()) {
        for (size_t i = 0; i != vec.size(); ++i) {
            elems[i] = vec[i];
        }
    }
};
```

Expression Templates (in a nutshell) - II

```
template <typename E1, typename E2>
class VecSum : public VecExpression<VecSum<E1, E2> > {
    E1 const& _u;
    E2 const& _v;

 public:
    VecSum(E1 const& u, E2 const& v) : _u(u), _v(v) {
        assert(u.size() == v.size());
    }

    double operator[](size_t i) const { return _u[i] + _v[i]; }
    size_t size()              const { return _v.size(); }
};

 template <typename E1, typename E2>
 VecSum<E1,E2> const
 operator+(E1 const& u, E2 const& v) {
    return VecSum<E1, E2>(u, v);
 }
```

```
Vec a, b, c;

auto Sum = a + b + c;

assert(typeof(sum) ==
VecSum<VecSum<Vec, Vec>, Vec>)

// evaluation only happens on
// assignment to type Vec

Vec evaluate = Sum;
```

“Modern C++” -- boost::ublas and Expression Templates

Roman Gareev

1. Detect operations on tropical semi-rings
   - SGEMM/DGEMM (+, *)
   - Floyd-Warshall (min, +)

2. Apply GOTO Algorithm (1)
   - L2 Tiling
   - Cache Transposed Submatrices
   - Register Tiling

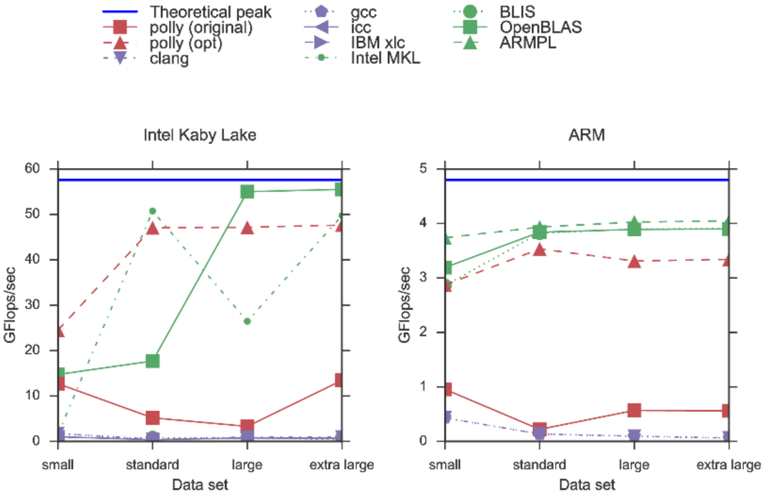
3. Chose optimal Cache and Register Tile Sizes (2)

Data-Layout Transformations in Polly

TargetData:
- L1/L2 Cache Sizes
- L2/L2 Cache Latencies

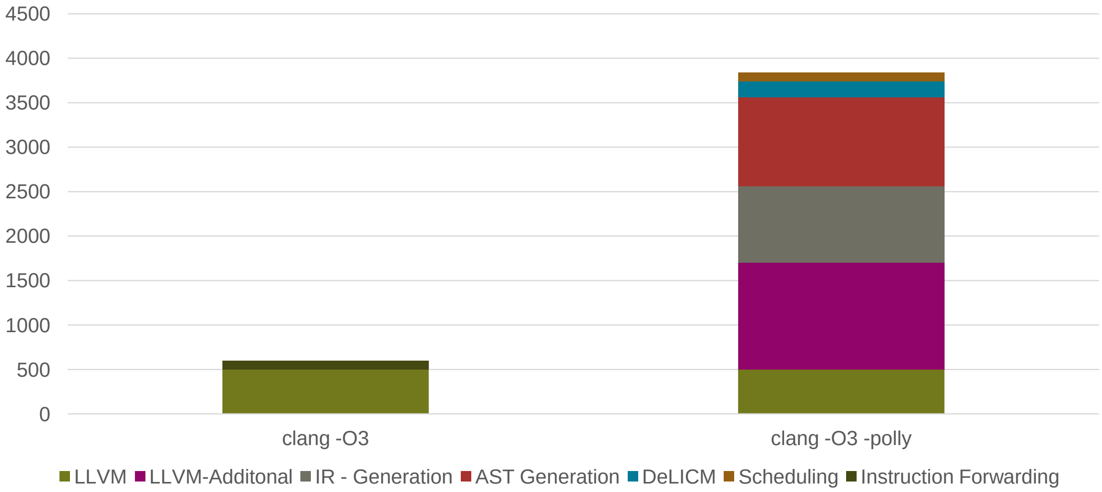
(1) High-performance implementation of the level-3 BLAS, Goto et al
(2) A Analytical Modeling is Enough for High Performance BLIS, Tzem Low et al

DGEMM Performance



Thanks @ARMHPC (Florian Hahn) for ARM codegen improvements!

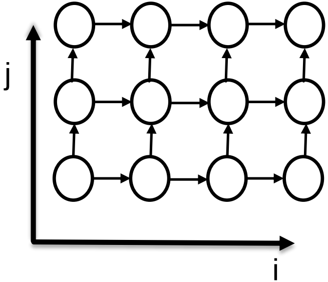
3MM Compile Time



"Provable" Correct Types for Loop Transformations

Maximilian Falkenstein

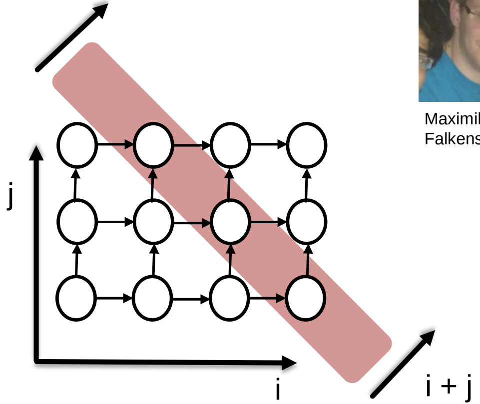
```
for (int32 i = 1; i < N; i++)
  for (int32 j = 1; j <= M; j++)
    A(i,j) = A(i-1,j) + A(i,j-1)
```



"Provable" Correct Types for Loop Transformations

```
for (int32 i = 1; i < N; i++)
  for (int32 j = 1; j <= M; j++)
    A(i,j) = A(i-1,j) + A(i,j-1)
```

```
for (intX c = 2; c < N+M; c++)
  #pragma simd
  for (intX i = max(1, c-M); i <= min(N, c-1); i++)
    A(i,c-i) = A(i-1,c-1) + A(i,c-i-1)
```

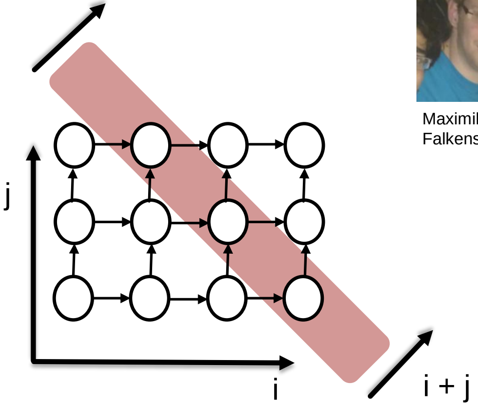


Maximilian Falkenstein

“Provable” Correct Types for Loop Transformations

Maximilian
Falkenstein

```
for (int32 i = 1; i < N; i++)
  for (int32 j = 1; j <= M; j++)
    A(i,j) = A(i-1,j) + A(i,j-1)
```



What is X?

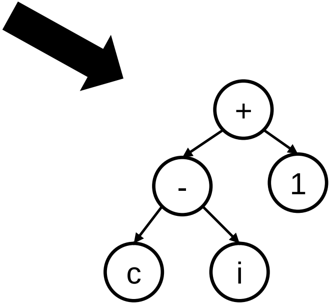
```
for (intX c = 2; c < N+M; c++)
  #pragma simd
  for (intX i = max(1, c-M); i <= min(N, c-1); i++)
    A(i,c-i) = A(i-1,c-1) + A(i,c-i-1)
```

Precise Solution

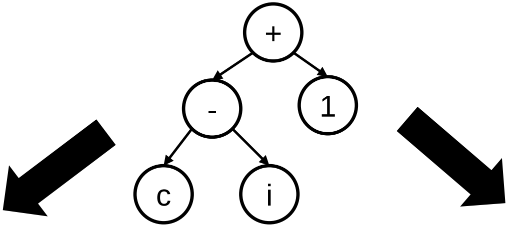
```
for (intX c = 2; c < N+M; c++)
  # simd
  for (intX i = max(1, c-M); i <= min(N, c-1); i++)
    A(i, c-i) = A(i-1, c-1) + A(i, c-i-1)
```

Domain: { [c] : 2 <= c < N + M
INT_MIN <= N, M <= INT_MAX }

f0() = c - i
f1() = c - i - 1



1) **calc:** min(fX()), max(fX()) under **Domain**
2) **choose type accordingly**



- Do you still target 32 bit?
- GPUs are faster in 32 bit
- FPGA?!

ILP Solver

- Minimal Types
- Potentially Costly

Approximations*

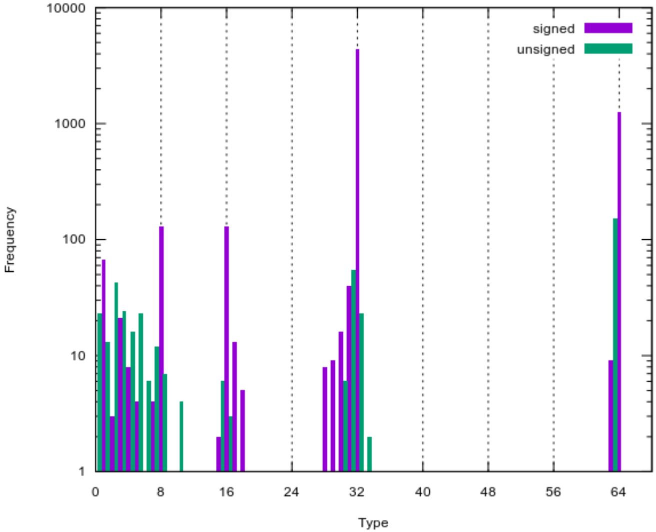
- $s(a+b) \leq \max(s(a), s(b)) + 1$
- Good, if smaller than native type

* Earlier uses in GCC and Polly

Preconditions

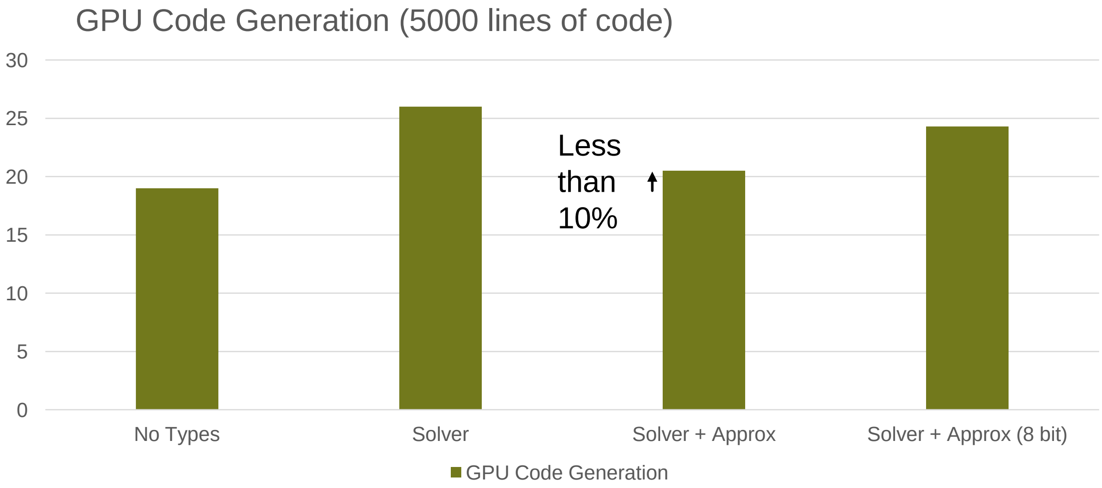
- Assume values fit into 32 bit
- Derive required pre-conditions

Type Distribution for LNT SCOPS



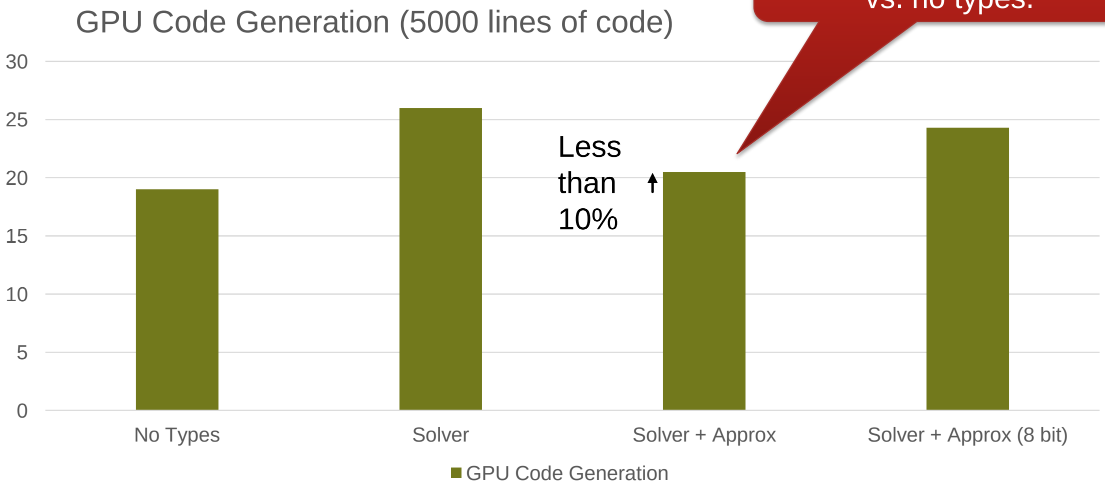
32 + epsilon is almost always enough!

Compile Time Overhead



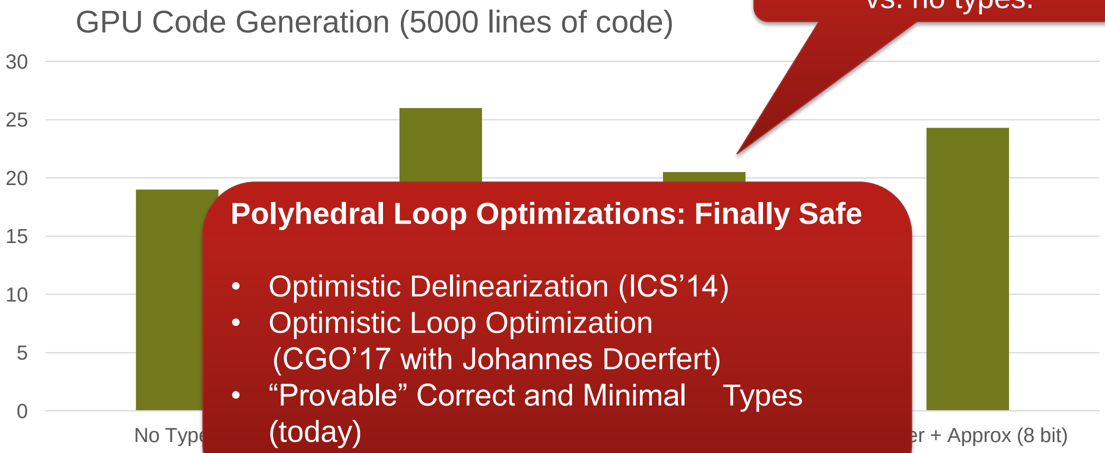
Compile Time Overhead

Less than 10% overhead vs. no types.



Compile Time Overhead

Less than 10% overhead vs. no types.



Polyhedral Loop Optimizations: Finally Safe

- Optimistic Delinearization (ICS'14)
- Optimistic Loop Optimization (CGO'17 with Johannes Doerfert)
- "Provable" Correct and Minimal Types (today)

Scalar Dependencies

Virtual Registers and PHI-Nodes

```
   for (int i=0; i<N; i++) {
S:   A[i] = ...;
T:   ... = A[i];
   }
```

- T(i) depends on S(i) — Read-After-Write/Flow-dependency
- S(0), S(1), …, S(N-1), T(0), T(1), … is a valid execution
- Parallel loop (OpenMP, OpenCL/PTX, tiling, vectorization, etc.)

Scalar Dependencies

Virtual Registers and PHI-Nodes

```
   for (int i=0; i<N; i++) {
S:   A[i] = ...;
T:   ... = A[i];
   }
```

```
   for (int i=0; i<N; i++) {
S:   tmp = ...;
T:   ... = tmp;
   }
```

"0-dimensional array"

Scalar Dependencies

Virtual Registers and PHI-Nodes

```
   for (int i=0; i<N; i++) {
S:   A[i] = ...;
T:   ... = A[i];
   }
```

```
   for (int i=0; i<N; i++) {
S:   tmp = ...;
T:   ... = tmp;
   }
```

"0-dimensional array"

- S(i) "depends" on S(i-1) — Write-After-Write/Output-dependency
- S(i) "depends" on T(i-1) — Write-After-Read/Anti-dependency
- S(0), S(1), …, T(0), … is **no** valid execution

Scalar Dependencies

Virtual Registers and PHI-Nodes

```
   for (int i=0; i<N; i++) {
S:   A[i] = ...;
T:   ... = A[i];
   }
```

mem2reg/SROA

```
   for (int i=0; i<N; i++) {
S:   tmp = ...;
T:   ... = tmp;
   }
```

"0-dimensional array"

- S(i) "depends" on S(i-1) — Write-After-Write/Output-dependency
- S(i) "depends" on T(i-1) — Write-After-Read/Anti-dependency
- S(0), S(1), …, T(0), … is **no** valid execution

Loop-Invariant Code Motion

```
   for (int i = 0; i < N; i += 1)
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i] * B[k];
```

Loop-Invariant Code Motion

```
   for (int i = 0; i < N; i += 1)
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i] * B[k];
```

⇓

```
   for (int i = 0; i < N; i += 1) {
T:   double tmp = A[i];
     for (int k = 0; k < K; k += 1)
S:     C[i] += tmp * B[k];
   }
```

Loop-Invariant Code Motion

```
    for (int i = 0; i < N; i += 1)
      for (int k = 0; k < K; k += 1)
S:      C[i] += A[i] * B[k];
```

GVN/LICM

```
    for (int i = 0; i < N; i += 1) {
T:    double tmp = A[i];
      for (int k = 0; k < K; k += 1)
S:      C[i] += tmp * B[k];
    }
```

Scalar Promotion in Loops

```
    for (int i = 0; i < N; i += 1) {
T:    C[i] = 0;
      for (int k = 0; k < K; k += 1)
S:      C[i] += A[i][k];
    }
```

Scalar Promotion in Loops

```
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i][k];
   }
```

```
   for (int i = 0; i < N; i += 1) {
T:   double tmp = 0;
     for (int k = 0; k < K; k += 1)
S:     tmp += A[i][k];
U:   C[i] = tmp;
   }
```

Scalar Promotion in Loops

```
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i][k];
   }
```

LICM

```
   for (int i = 0; i < N; i += 1) {
T:   double tmp = 0;
     for (int k = 0; k < K; k += 1)
S:     tmp += A[i][k];
U:   C[i] = tmp;
   }
```

Speculative Execution

```
   for (int i = 0; i < N; i += 1) {
     if (i > 5)
S1:    C[i] = 5 + 2*x;
     else
S2:    C[i] = 7 + 2*x;
   }
```

Speculative Execution

```
    for (int i = 0; i < N; i += 1) {
      if (i > 5)
S1:     C[i] = 5 + 2*x;
      else
S2:     C[i] = 7 + 2*x;
    }
```

EarlyCSE/GVN/NewGVN/GVNHoist

```
    for (int i = 0; i < N; i += 1) {
T:    double tmp = 2*x;
      if (i > 5)
S1:     C[i] = 5 + tmp;
      else
S2:     C[i] = 7 + tmp;
    }
```

(Partial) Redundancy Elimination

```
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i][k];
   }
```

(Partial) Redundancy Elimination

```
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i][k];
   }
```

GVN Load PRE

```
   for (int i = 0; i < N; i += 1) {
T:   double tmp = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] = (tmp += A[i][k]);
   }
```

Loop Idiom

“doitgen” – Multiresolution Kernel from MADNESS

```
for (int r = 0; r < R; r++)
  for (int q = 0; q < Q; q++) {
    for (int p = 0; p < P; p++) {
      sum[p] = 0;
      for (int s = 0; s < P; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < P; p++)
      A[r][q][p] = sum[p];
  }
```

Loop Idiom

“doitgen” – Multiresolution Kernel from MADNESS

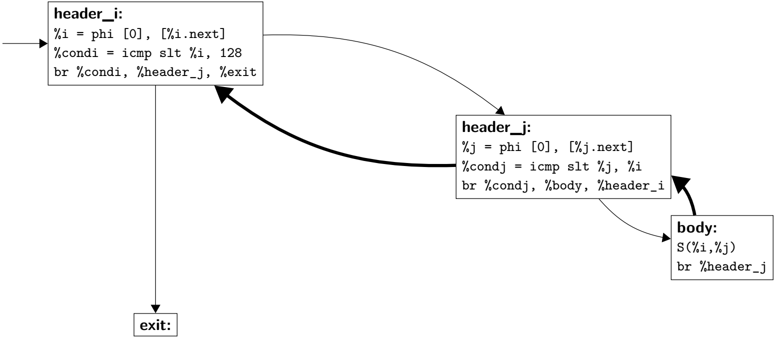
```
for (int r = 0; r < R; r++)
  for (int q = 0; q < Q; q++) {
    for (int p = 0; p < P; p++) {
      sum[p] = 0;
      for (int s = 0; s < P; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < P; p++)
      A[r][q][p] = sum[p];
  }
```

LoopIdiom

```
for (int r = 0; r < R; r++)
  for (int q = 0; q < Q; q++) {
    for (int p = 0; p < P; p++) {
      sum[p] = 0;
      for (int s = 0; s < P; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    memcpy(A[r][q], sum, sizeof(sum[i]) * P);
  }
```

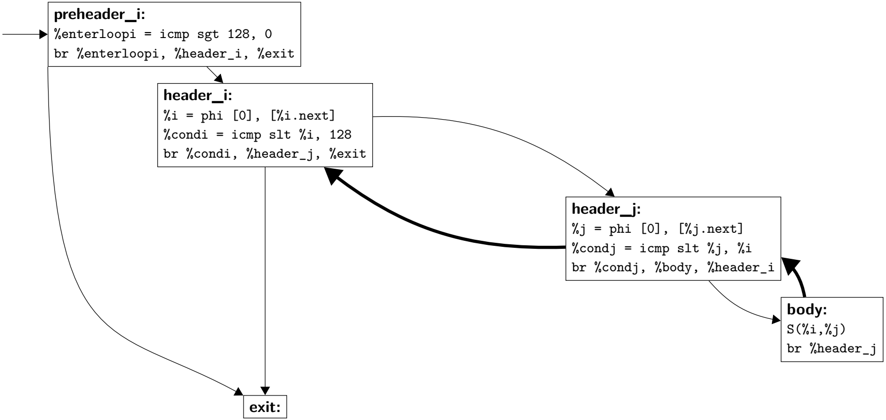
Loop Rotation

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



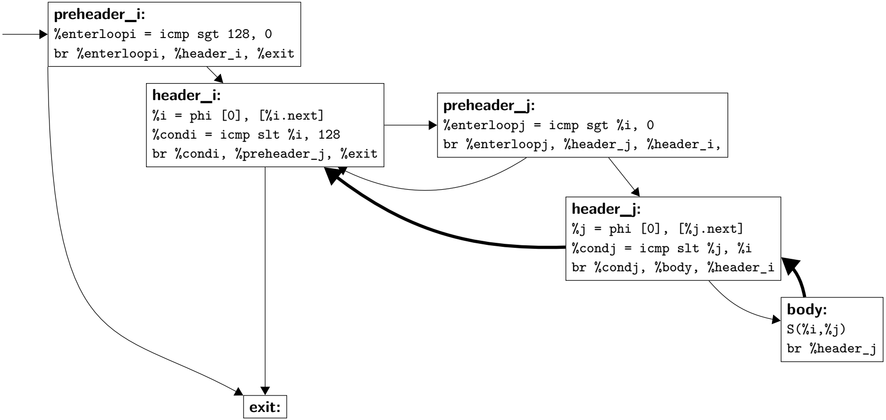
Loop Rotation

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



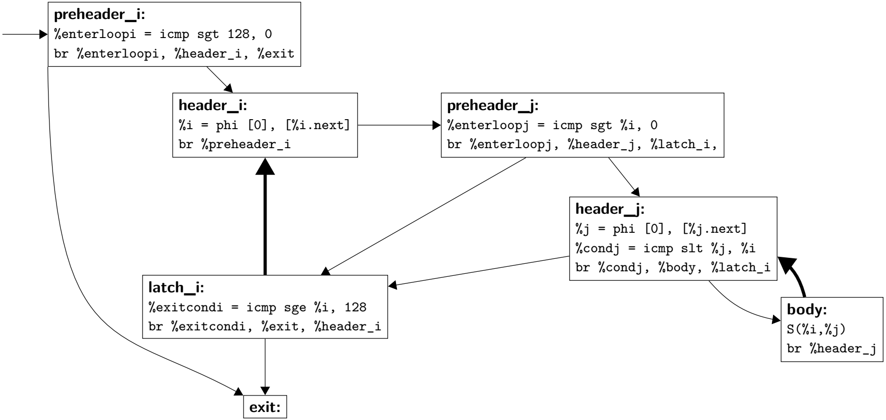
Loop Rotation

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



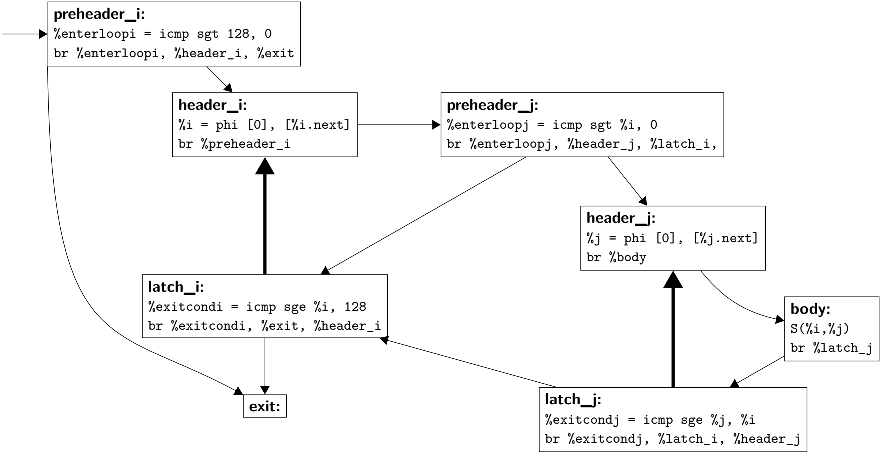
Loop Rotation

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



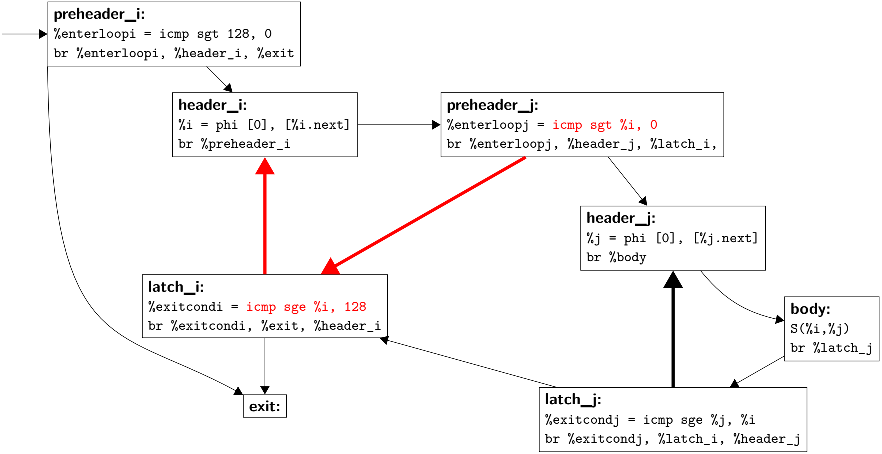
Loop Rotation

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



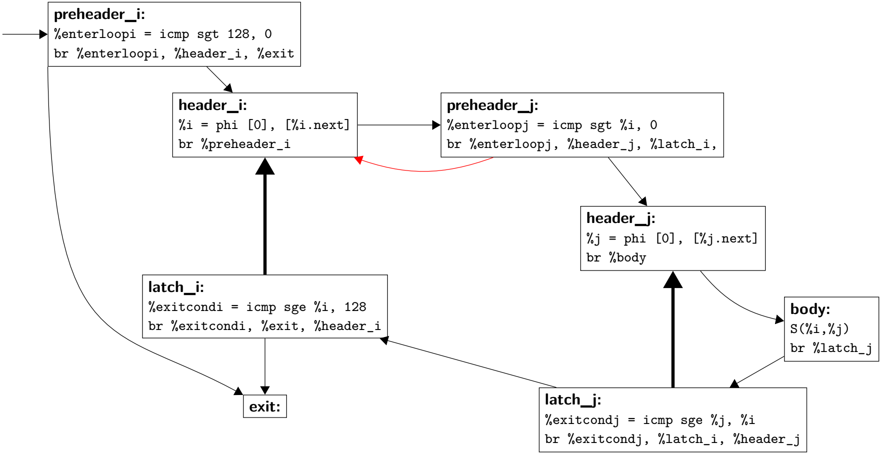
Jump Threading

```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



Jump Threading

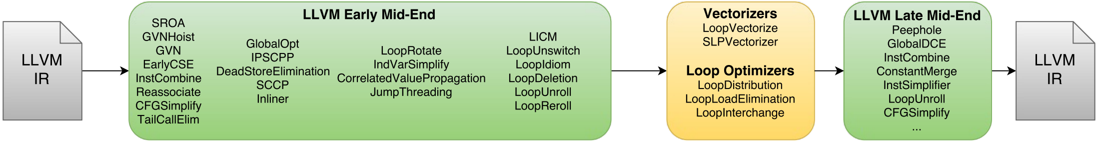
```
for (int i = 0; i < 128; i += 1)
  for (int j = 0; j < i; j += 1)
    S(i,j);
```



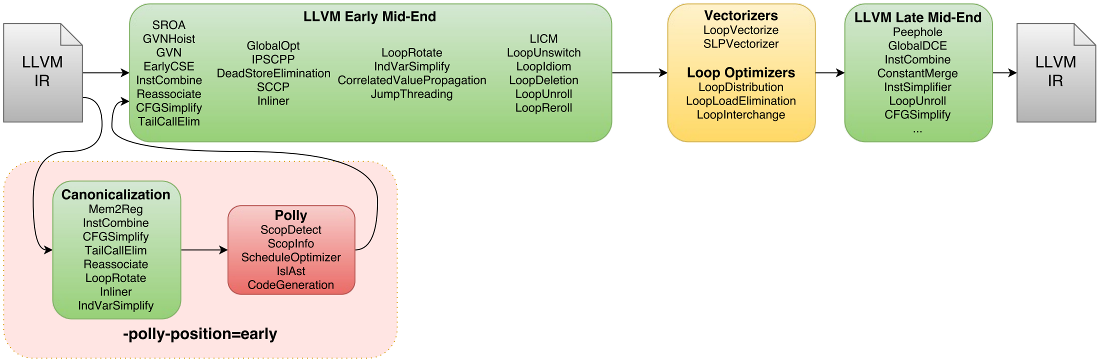
Chapter Summary

- Semantically identical IR can be harder to optimize
- Possible causes:
  - Static Single Assignment form (e.g. mem2reg)
  - Non-Polyhedral transformation passes (e.g. GVN, LICM)
  - C++ abstraction layers (e.g. Boost uBLAS)
  - Manual source optimizations (e.g. loop hoisting)
  - Code generators (e.g. TensorFlow XLA)

LLVM Pass Pipeline -O3

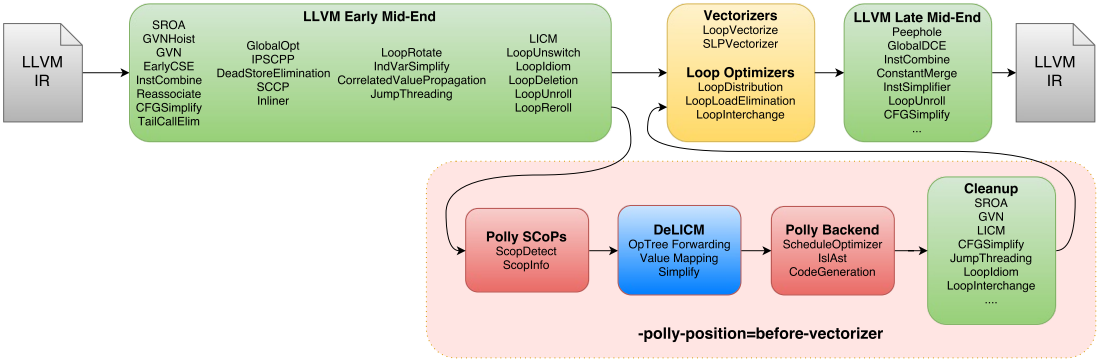


LLVM Pass Pipeline -O3 -polly -polly-position=early

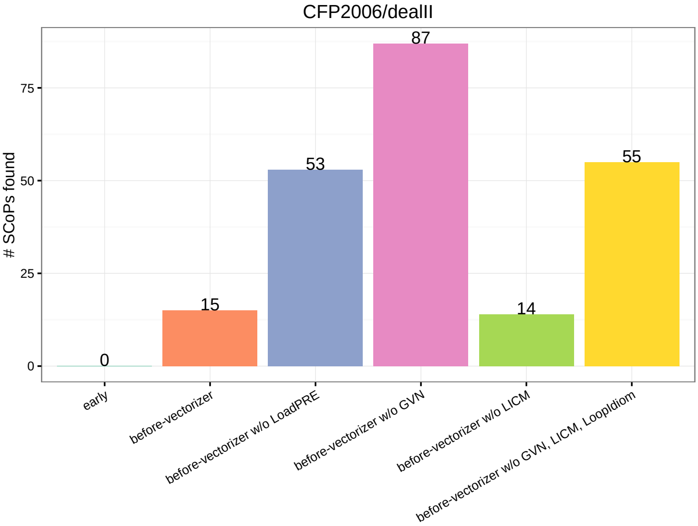


LLVM Pass Pipeline

-O3 -polly -polly-position=before-vectorizer



Effects of GVN, LICM, LoopIdiom



Value Mapping

"DeLICM"

```
   double c;
   for (int i = 0; i < 3; i += 1) {
T:   c = 0;
     for (int k = 0; k < 3; k += 1)
S:     c += A[i] * B[k];
U:   C[i] = c;
   }
```

Value Mapping

“DeLICM”

```
    double c;
    for (int i = 0; i < 3; i += 1) {
T:    c = 0;
      for (int k = 0; k < 3; k += 1)
S:      c += A[i] * B[k];
U:    C[i] = c;
    }
```

	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$	$U(0)$	$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$	$U(1)$	$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$	$U(2)$	
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Value Mapping “DeLICM”

```
   double c;
   for (int i = 0; i < 3; i += 1) {
T:   c = 0;
     for (int k = 0; k < 3; k += 1)
S:     c += A[i] * B[k];
U:   C[i] = c;
   }
```

	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$	$U(0)$	$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$	$U(1)$	$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$	$U(2)$	
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

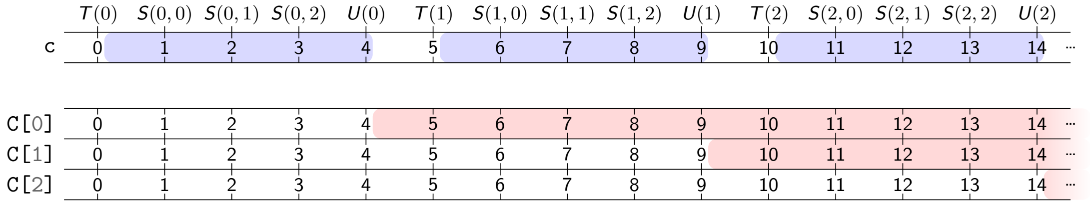
Value Mapping "DeLICM"

```
    double c;
    for (int i = 0; i < 3; i += 1) {
T:    c = 0;
      for (int k = 0; k < 3; k += 1)
S:      c += A[i] * B[k];
U:    C[i] = c;
    }
```

	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$	$U(0)$	$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$	$U(1)$	$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$	$U(2)$	
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[1]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[2]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Value Mapping "DeLICM"

```
    double c;
    for (int i = 0; i < 3; i += 1) {
T:    c = 0;
      for (int k = 0; k < 3; k += 1)
S:      c += A[i] * B[k];
U:    C[i] = c;
    }
```



Value Mapping "DeLICM"

```
   double c;
   for (int i = 0; i < 3; i += 1) {
T:   C[2] = 0;
     for (int k = 0; k < 3; k += 1)
S:     C[2] += A[i] * B[k];
U:   C[i] = C[2];
   }
```

	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$	$U(0)$	$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$	$U(1)$	$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$	$U(2)$	
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[1]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
C[2]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Value Mapping

“DeLICM”

```
   double c;
   for (int i = 0; i < 3; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < 3; k += 1)
S:     C[i] += A[i] * B[k];
U:   C[i] = C[i];
   }
```

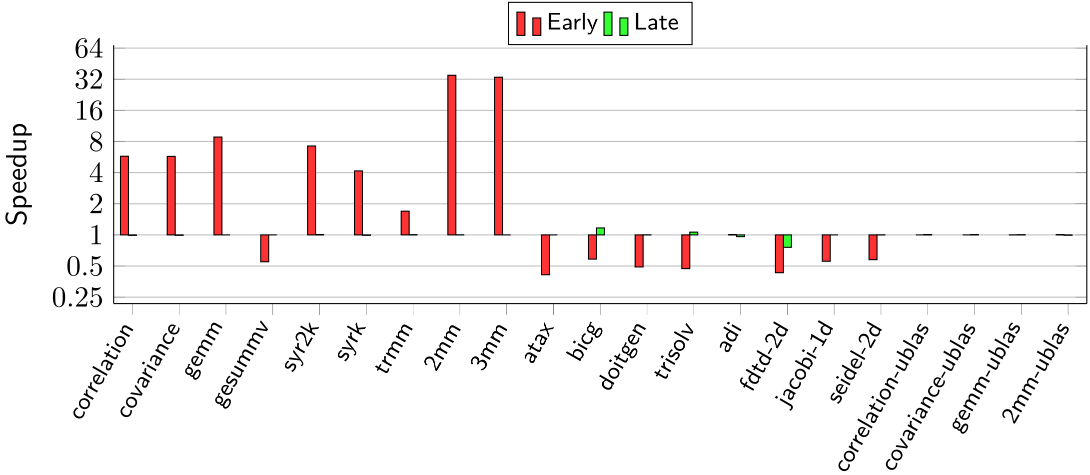
	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$	$U(0)$	$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$	$U(1)$	$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$	$U(2)$	
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[1]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[2]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Value Mapping “DeLICM”

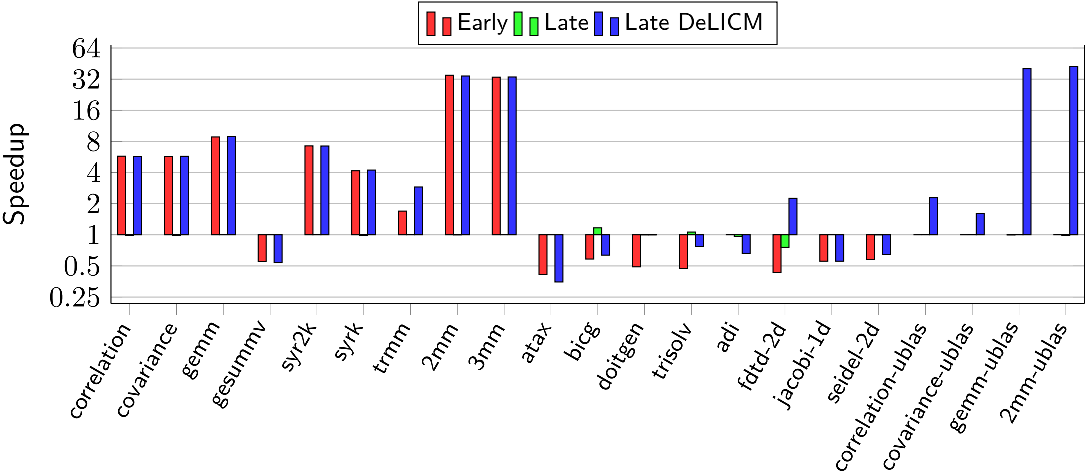
```
    double c;
    for (int i = 0; i < 3; i += 1) {
T:    C[i] = 0;
      for (int k = 0; k < 3; k += 1)
S:      C[i] += A[i] * B[k];
    }
```

	$T(0)$	$S(0,0)$	$S(0,1)$	$S(0,2)$		$T(1)$	$S(1,0)$	$S(1,1)$	$S(1,2)$		$T(2)$	$S(2,0)$	$S(2,1)$	$S(2,2)$		
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[1]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
C[2]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Experiments



Experiments



Chapter Summary

- LLVM mid-end canonicalization inhibits polyhedral optimization
- Can undo scalar optimizations on the polyhedral representation ("DeLICM")
- Reasons to run Polly after canonicalization:
  - More optimizations, especially the inliner
  - More canonicalized, less dependent on input code
  - Avoid running canonicalization passes redundantly
  - No IR-modification when no polyhedral transformation was done

SPEC CPU 2006 456.hmmer

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

SPEC CPU 2006 456.hmmer

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

SPEC CPU 2006 456.hmmer

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

Compute `mc[k]` (*vectorizable*)

Compute `dc[k]` (**not** vectorizable)

Compute `ic[k]` (*vectorizable*)

LoopDistribution/LoopVectorizer -enable-loop-distribute

- Gerolf Hoflehner, *LLVM Performance Improvements and Headroom*, LLVM DevMtg 2015

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
}
```

loop-vectorized

```
for (k = 1; k <= M; k++) {
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

not vectorized

LoopDistribution/LoopVectorizer

-loop-distribute-non-if-convertible

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
}
```

loop-vectorized

```
for (k = 1; k <= M; k++) {
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
}
```

not vectorized

```
for (k = 1; k <= M; k++) {
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

vectorized with if-conversion

Polly -polly-stmt-granularity=bb

```
      for (k = 1; k <= M; k++) {
Stmt1:  mc[k] = mpp[k-1]   + tpmm[k-1];
        if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
        if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
        if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
        mc[k] += ms[k];
        if (mc[k] < -INFTY) mc[k] = -INFTY;
        dc[k] = dc[k-1] + tpdd[k-1];
        if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
        if (dc[k] < -INFTY) dc[k] = -INFTY;

        if (k < M) {
Stmt2:    ic[k] = mpp[k] + tpmi[k];
          if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
          ic[k] += is[k];
          if (ic[k] < -INFTY) ic[k] = -INFTY;
        }
      }
```

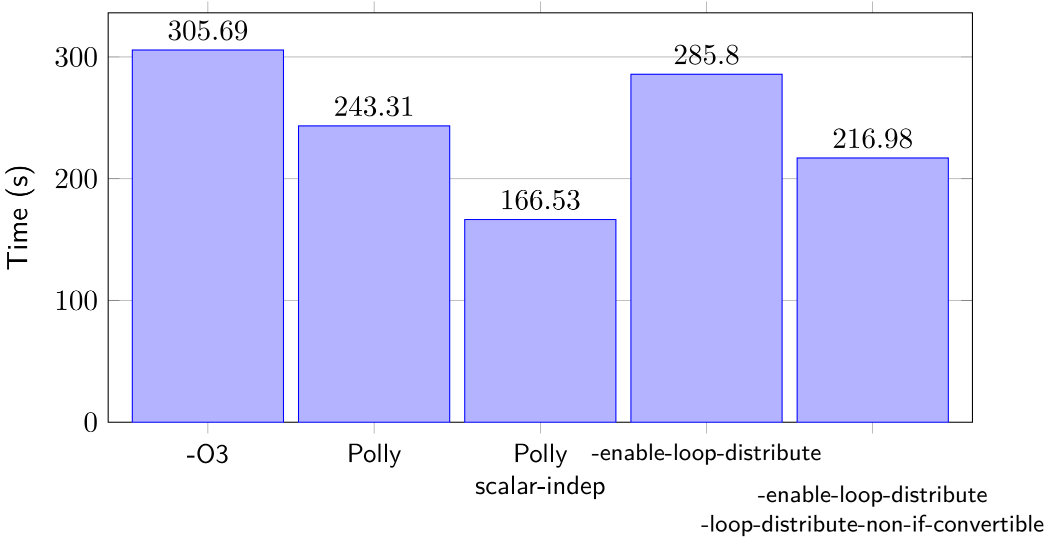
Polly -polly-stmt-granularity=scalar-indep

```
    for (k = 1; k <= M; k++) {
Stmt1: mc[k] = mpp[k-1]   + tpmm[k-1];
       if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
       if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
       if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
       mc[k] += ms[k];
       if (mc[k] < -INFTY) mc[k] = -INFTY;

Stmt2: dc[k] = dc[k-1] + tpdd[k-1];
       if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
       if (dc[k] < -INFTY) dc[k] = -INFTY;

       if (k < M) {
Stmt3:   ic[k] = mpp[k] + tpmi[k];
         if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
         ic[k] += is[k];
         if (ic[k] < -INFTY) ic[k] = -INFTY;
       }
    }
```

Loop Distribution by Polyhedral Scheduler

```
$ opt -polly-stmt-granularity=scalar-indep -polly-invariant-load-hoisting -polly-use-llvm-names \
  fast_algorithms.ll -polly-opt-isl -polly-ast -analyze

[...]

    {
      for (int c0 = 0; c0 < _lcssa; c0 += 1)
        Stmt_for_body72(c0);
      for (int c0 = 0; c0 < _lcssa; c0 += 1)
        Stmt_for_body721(c0);
      for (int c0 = 0; c0 < _lcssa - 1; c0 += 1)
        Stmt_if_then167(c0);
      if (_lcssa >= 1)
        Stmt_for_end204_loopexit();
    }
```

Experiments



Chapter Summary

- Finer-grained statements
- One basic block => multiple statement if no computation is shared
- Enables loop distribution by Polly
- Speed-up of 80% in 456.hmmer

- With support from Nandini Singhal

That's all Folks!

Summary

1. COSMO weather forecasting on GPGPUs
2. Life Range Reordering (Verdoolaege IMPACT'16)
3. DGEMM detection also with C++ expression templates (Roman Gareev)
4. Correct types for loop transformations (Maximilian Falkenstein)
5. Some LLVM passes make polyhedral optimization harder
6. `-polly-position=early` vs. `-polly-position=before-vectorizer`
7. DeLICM: Avoiding scalar dependencies
8. `-polly-stmt-granularity` and loop-distribution in 456.hmmer (with Nandini Singhal)