Polly’s Polyhedral Scheduling in the Presence of Reductions

Johannes Doerfert*
Kevin Streit*
Sebastian Hack*
Zino Benaissa†

* Saarland University
Saarbrücken, Germany
† Qualcomm Innovation Center
San Diego, USA

January 19, 2015
for (i = 0; i < 4 * N; i++)
    sum += A[i];

Reductions

\[
tmp\_sum[4] = \{0, 0, 0, 0\} \\
for (i = 0; i < 4 * N; i+=4) \\
    tmp\_sum[0:3] += A[i:i+3]; \\

sum += tmp\_sum[0] + tmp\_sum[1]; \\
    + tmp\_sum[2] + tmp\_sum[3];
\]

for (i = 0; i < 4 * N; i++) {
    S(i);
    sum += A[i];
    P(i);
}
Reductions

tmp_sum[4] = {0,0,0,0}

for (i = 0; i < 4 * N; i+=4) {
    vecS(i:i+3);
    tmp_sum[0:3] += A[i:i+3];
    vecP(i:i+3);
}

sum += tmp_sum[0] + tmp_sum[1];

Reductions

```c
for (i = 0; i < NX; i++) {
    for (j = 0; j < NY; j++) {
        q[i] = q[i] + A[i][j] * p[j];
        s[j] = s[j] + r[i] * A[i][j];
    }
}
```


Reductions

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

for (i = N; i >= 0; i--)
    sum += A[i];
Reductions

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

sums[N+1] = sum;
for (i = N; i >= 0; i--)
    sums[i] = sums[i+1] + A[i];
sum = sums[0];

Reductions

```c
sums[N+1] = sum;
for (i = 0; i <= N; i++) {
    A[i] = i;
    sums[i] = sums[i+1] + A[i];
}
sum = sums[0];
```

Reductions

SCoP Detection

Precise Dependence Analysis + Scalar/Array Expansion

Reduction-enabled SRE Scheduling

Code Generation

Storage Mapping
Objectives & Challenges

Objectives

1) Detect general reduction computations
2) Parallelize/Vectorize reductions efficiently
3) Interchange the order reductions are computed

Practical Considerations

a) Avoid runtime regressions
b) Minimize memory overhead
c) Minimize compile time overhead
Objectives & Challenges

Objectives

1) Detect general reduction computations
2) Parallelize/Vectorize reductions efficiently
3) Interchange the order reductions are computed

Practical Considerations

a) Avoid runtime regressions
b) Minimize memory overhead
c) Minimize compile time overhead
Objectives & Challenges

Objectives

1) Detect general reduction computations
2) Parallelize/Vectorize reductions efficiently
3) Interchange the order reductions are computed

Practical Considerations

a) Avoid runtime regressions
b) Minimize memory overhead
c) Minimize compile time overhead
Overview — Polly in LLVM
Overview — Polly in LLVM

LLVM-IR → SCoP Detection → Polyhedral Dependence Analysis → Polyhedral Scheduling → Polyhedral Code Generation → LLVM-IR
Overview — Polly in LLVM

- LLVM-IR
  - SCoP Detection
  - Reduction-like Computation Detection
  - Polyhedral Dependence Analysis
  - Polyhedral Scheduling
  - Polyhedral Code Generation
  - LLVM-IR
Reduction-like Computations

- Updates on the same memory cells
- Associative & commutative computations
- Locally not observed or intervened
Reduction-like Computations

- Updates on the same memory cells
- Associative & commutative computations
- Locally not observed or intervened

Details are provided in the paper.
Overview — Polly in LLVM

- LLVM-IR
- SCoP Detection
- Reduction-like Computation Detection
- Polyhedral Dependence Analysis
- Polyhedral Scheduling
- Polyhedral Code Generation
- LLVM-IR
Overview — Polly in LLVM

LLVM-IR → SCoP Detection → Reduction-like Computation Detection → Reduction-aware Polyhedral Dependence Analysis → Polyhedral Scheduling → Polyhedral Code Generation → LLVM-IR
Reduction Dependences

- Loop carried self dependences
- Induced by reduction-like computations
- Inherit “associative” & “commutative” properties

Reduction Dependences

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
        ...
        sum += A[i];
        ...
    }
    return sum;
}
```

Dependence Analysis

- Performed on statement level
- Computes value-based dependences
Reduction Dependences

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
        {
            S: ...;
            R: sum += A[i];
            S: ...
        }
    return sum;
}
```

Dependence Analysis
- Performed on statement level
- Computes value-based dependences

Reduction Dependence Analysis
- Isolates the load & store of reduction-like computations
- Performed both on access and statement level
- Identifies reuse of values by a reduction-like computation
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
        sum += A[i];
    return sum;
}

Dependences
{Stmt_S[i0] → Stmt_S[1+i0]: i0 >= 0 and i0 <= N - 1}
Reduction Dependences

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
        R: sum += A[i];
    return sum;
}
```

Dependences

```
{
}
```

Reduction Dependences

```
{Stmt_R[i0] → Stmt_R[1+i0]: i0 >= 0 and i0 <= N − 1}
```
Reduction Dependences

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
    {
        sum += i;
    }
    return sum;
}
```

Dependences

\{ Stmt.S[i0] \rightarrow Stmt.S[1+i0] : i0 \geq 0 \text{ and } i0 \leq N - 1 \}
Reduction Dependences

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
    {
        R: sum += i;
    }
    return sum;
}
```

Dependences

\{ Stmt_S[i0] \rightarrow Stmt_S[1+i0] : i0 >= 0 and i0 <= N-1 \}

Reduction Dependences

\{ Stmt_R[i0] \rightarrow Stmt_R[1+i0] : i0 >= 0 and i0 <= N-1 \}
void bicg(float q[NX], ...) {
    for (int i = 0; i < NX; i++) {
        S: q[i] = 0;
        for (int j = 0; j < NY; j++)
            T: {
                q[i] = q[i] + A[i][j] * p[j];
                s[j] = s[j] + r[i] * A[i][j];
            }
    }
}
void bicg(float q[NX], ...) {
    for (int i = 0; i < NX; i++) {
        S: q[i] = 0;
        for (int j = 0; j < NY; j++) {
            R1: q[i] = q[i] + A[i][j] * p[j];
            R2: s[j] = s[j] + r[i] * A[i][j];
        }
    }
}
Overview — Polly in LLVM

1. LLVM-IR → SCoP Detection
2. SCoP Detection → Reduction-like Computation Detection
3. Reduction-like Computation Detection → Reduction-aware Polyhedral Dependence Analysis
4. Reduction-aware Polyhedral Dependence Analysis → Polyhedral Scheduling
5. Polyhedral Scheduling → Polyhedral Code Generation
6. Polyhedral Code Generation → LLVM-IR
Overview — Polly in LLVM

 LLVM-IR → SCoP Detection → Reduction-like Computation Detection → Reduction-aware Polyhedral Dependence Analysis → Polyhedral Scheduling → Reduction Modeling → Polyhedral Code Generation → LLVM-IR
Reduction Modeling
Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check *with* and *without* reduction dependences
Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check *with* and *without* reduction dependences

Reduction-enabled Scheduling

- Ignore reduction dependences during the scheduling
- May need additional *privatization dependences*
Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check with and without reduction dependences

Reduction-enabled Scheduling

- Ignore reduction dependences during the scheduling
- May need additional privatization dependences

Reduction-aware Scheduling

- Let the scheduler make the parallelization decision based on the environment and the potential cost of privatization
Overview — Polly in LLVM

LLVM-IR → SCoP Detection → Reduction-like Computation Detection → Reduction-aware Polyhedral Dependence Analysis → Reduction-enabled Polyhedral Code Generation → Polyhedral Scheduling → Reduction Modeling → LLVM-IR
Overview — Polly in LLVM

LLVM-IR → SCoP Detection → Reduction-like Computation Detection → Reduction-aware Polyhedral Dependence Analysis → Reduction-enabled Polyhedral Code Generation → Reduction-enabled Polyhedral Scheduling → Reduction Modeling → LLVM-IR
Reduction-enabled Scheduling

```c
void bicg(float q[NX], ...) {
    for (int i = 0; i < NX; i++) {
        S: q[i] = 0;
        for (int j = 0; j < NY; j++)
            R1: q[i] = q[i] + A[i][j] * p[j];
        R2: s[j] = s[j] + r[i] * A[i][j];
    }
}
```

Dependences

```c
{Stmt_S[i0] → Stmt_R1[i0, 0]: i0 >= 0 and i0 <= NX}
```

Reduction Dependences

```c
{Stmt_R1[i0, i1] → Stmt_R1[i0, 1 + i1]:...}
{Stmt_R2[i0, i1] → Stmt_R2[1 + i0, i1]:...}
```
Reduction-enabled Scheduling

Privatization Dependences

- Transitive extension along reduction dependences
- Already contained in memory based dependences
- Order reduction computations and others on the same memory cells
Reduction-enabled Scheduling

```c
void bicg(float q[NX], ...) {
    for (int i = 0; i < NX; i++) {
        S: q[i] = 0;
        for (int j = 0; j < NY; j++)
            R1: q[i] = q[i] + A[i][j] * p[j];
            R2: s[j] = s[j] + r[i] * A[i][j];
    }
}
```

Dependences

\{Stmt.S[i0] \rightarrow Stmt.R1[i0, 0] : i0 \geq 0 \text{ and } i0 \leq NX\}

Reduction Dependences

\{Stmt.R1[i0, i1] \rightarrow Stmt.R1[i0, 1 + i1] : \ldots \}
\{Stmt.R2[i0, i1] \rightarrow Stmt.R2[1 + i0, i1] : \ldots \}\
Reduction-enabled Scheduling

```c
void bicg(float q[NX], ...) {
    for (int i = 0; i < NX; i++) {
        S: q[i] = 0;
        for (int j = 0; j < NY; j++) {
            R1: q[i] = q[i] + A[i][j] * p[j];
            R2: s[j] = s[j] + r[i] * A[i][j];
        }
    }
}
```

Dependences

\{ Stmt_S[i0] \rightarrow Stmt_R1[i0, 0] : i0 \geq 0 \text{ and } i0 \leq NX \}\}

Reduction Dependences

\{ Stmt_R1[i0, i1] \rightarrow Stmt_R1[i0, 1 + i1] : \ldots \}\}
\{ Stmt_R2[i0, i1] \rightarrow Stmt_R2[1 + i0, i1] : \ldots \}\}

Privatization Dependences

\{ Stmt_S[i0] \rightarrow Stmt_R1[i0, o0] : o0 \geq 1 \text{ and } o0 \leq NY - 1 \text{ and } i0 \geq 0 \text{ and } i0 \leq NX \}\}

42/54
Evaluation — Compile Time
Evaluation — Compile Time

Statement-wise Dependence Analysis

- Standard value-based dependence analysis in Polly

Hybrid Dependence Analysis

- Adds 85% in average — takes up to 5× as long

Access-wise Dependence Analysis

- Adds \( \sim 170\% \) in average — takes up to 10× as long
Evaluation — Compile Time

Overhead vs. Statement-wise dependences

Access-wise dependences
Hybrid dependences

2mm
3mm
adi
atix
bicg
cholesky
correlation
covariance
doigten
durbin
dynprog
fdtd-2d
fdtd-apml
floyd-warshall
gemm
gemver
gesummv
gramschmidt
jacobi-1d-imper
jacobi-2d-imper
lu
ludcmp
mvt
reg-detect
dseidel-2d
symm
syrk
trisolv
trmm

45/54
Evaluation — Runtime
Evaluation — Runtime

Runtime Evaluation Notes

- Polly’s heuristic to choose a vector dimension is underdeveloped
- The LLVM vectorizer can treat simple innermost (scalar) reductions
- Polybench is highly parallel $\rightarrow$ reduction parallelism is almost never needed
void bind(Taint q(XX), ... )
{
    for (int i = 0; i < XX; i++) {
        q[i] = 0;
        for (int j = 0; j < YY; j++)
        {
            r1: q[i] = q[i] + a[i][j] + p[j];
            r2: s[j] = s[j] + r[i] + a[i][j];
        }
    }
}

Dependences:

{Stat.2[i]\rightarrow Stat.2[x0, \ldots] : x0 >= 0 and x0 <= XX}

Reduction Dependences:

{Stat.2[i][0, 1] \rightarrow Stat.2[i][0, 1 + 1]; ... }

{Stat.3[i][0, 1] \rightarrow Stat.3[i][1, 10, 1]; ... }

Privatization Dependences:

{Stat.2[i]\rightarrow Stat.3[i][10, 0] : x0 >= 1 and x0 <= 0 + x0 < YY - 1 and 10 >= 0 and 10 <= XX}
Conclusion

void hing(Tset q[X], ...) { 
    for (int i = 0; i < X; i++) { 
        q[i] = 0; 
        for (int j = 0; j < Y; j++) 
            q[i] += p[j]; 
    } 
} 

Dependences:

Reduction Dependences:

Privatization Dependences:
\{ Stat.A[0] \rightarrow Stat.A[10, 00] : i0 >= 1 and i0 <= Y - 1 and 10 >= 0 and 10 <= X \}
Conclusion

Dependences:
\{ \text{Stat.3}[c] \rightarrow \text{Stat.3}[10, c] : 10 \geq 0 \text{ and } 10 < \text{XX} \}

Reduction Dependences:
\{ \text{Stat.3}[10, 1] \rightarrow \text{Stat.3}[10, 1 + 1] \ldots \}
\{ \text{Stat.3}[10, 11] \rightarrow \text{Stat.3}[1 + 10, 11] \ldots \}

Privatization Dependences:
\{ \text{Stat.2}[c] \rightarrow \text{Stat.3}[10, c] : c0 < 1 \text{ and } c0 < \text{YY} - 1 \\
\text{and } 10 \geq 0 \text{ and } 10 < \text{XX} \}
void bing(int q[][], ...) { 
    for (int i = 0; i < N; i++) {
        if (i > 0) {
            for (int j = 0; j < N1; j++) {
                q[i][j] = q[i][j - 1] + p[q[1][j]]; 
            }
            for (int k = 0; k < N2; k++) {
                a[i][k] = a[i][k] + r[i][k]; 
            }
        }
    }
}

Dependences:
{Stat_a[0] \rightarrow Stat_a[10, 0] : 00 >= 0 and 10 <= N1}

Reduction Dependences:
{Stat_a[10, 11] \rightarrow Stat_a[10, 1 + 11] \ldots} 
{Stat_a[2][10, 11] \rightarrow Stat_a[2][1 + 10, 11] \ldots} 

Privatization Dependences:
{Stat_a[0] \rightarrow Stat_a[10, 0, 0] : 00 >= 1 and 10 <= N2 - 1 and 10 >= 0 and 10 <= N1}

---

Thank You.
Extensions

```c
for (i = 0; i < N; i++) {
    S(i);
    last = f(i);
}
```

Unary Reductions

- Induce only WAW dependences
- Can be reordered or parallelized
- Only the last value needs to be recovered
Extensions

```c
for (i = 0; i < N; i++) {
    sum += A[i];
    S(i);
    sum += B[i];
}
```

Multiple Statement Reductions

- Allowed between “compatible” reductions
- Induce dependence cycles, no self dependences
- Complicate efficient code generation/privatization
Extensions

for (i = 0; i < N; i++)

Scans/Recurrences

- Induce only RAW dependences
- Cannot be reordered but parallelized
- Different code generation than reductions
Reduction-like Computation Detection

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++)
        sum += A[i];
    return sum;
}
```

```assembly
define i32 @f(i32* %A, i32 %N) {
  entry:
    %sum = alloca i32
    store i32* %sum, i32 0
    br label %for.cond
  for.cond:
    %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
    %cmp = icmp slt i32 %iv, %N
    br i1 %cmp, label %for.body, label %for.end
  for.body:
    %idx = getelementptr inbounds i32* %A, i32 %iv
    %tmp1 = load i32* %idx, align 4
    %sum.reload = load i32* %sum
    %add = add nsw i32 %sum.reload, %tmp1
    store i32* %sum, i32 %add
    %iv.next = add nuw nsw i32 %iv, 1
    br label %for.cond
  for.end:
    %sum.reload3 = load i32* %sum
    ret i32 %sum.reload3
}
```
Reduction-like Computation Detection

```c
int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum += A[i];
        A[i] = sum;
    }
    return sum;
}
```

```mlang
define i32 @f(i32* %A, i32 %N) {
  entry:
    %sum = alloca i32
    store i32* %sum, i32 0
    br label %for.cond

  for.cond:
    %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
    %cmp = icmp slt i32 %iv, %N
    br i1 %cmp, label %for.body, label %for.end

  for.body:
    %idx = getelementptr inbounds i32* %A, i32 %iv
    %tmp1 = load i32* %idx, align 4
    %sum.reload = load i32* %sum
    %add = add nsw i32 %sum.reload, %tmp1
    store i32* %sum, i32 %add
    store i32* %idx, i32 %add
    %iv.next = add nuw nsw i32 %iv, 1
    br label %for.cond

  for.end:
    %sum.reload3 = load i32* %sum
    ret i32 %sum.reload3
}
```
Reduction-like Computation Detection

int f(int *A, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        int tmp = sum;
        sum += A[i];
        A[i] = tmp;
    }
    return sum;
}

define i32 @f(i32* %A, i32 %N) {
    entry:
    %sum = alloca i32
    store i32* %sum, i32 0
    br label %for.cond
    
for.cond:
    %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
    %cmp = icmp slt i32 %iv, %N
    br i1 %cmp, label %for.body, label %for.end

for.body:
    %idx = getelementptr inbounds i32* %A, i32 %iv
    %tmp1 = load i32* %idx, align 4
    %sum.reload = load i32* %sum
    %add = add nsw i32 %sum.reload, %tmp1
    store i32* %sum, i32 %add

    store i32* %idx, i32 %sum.reload
    %iv.next = add nuw nsw i32 %iv, 1
    br label %for.cond

for.end:
    %sum.reload3 = load i32* %sum
    ret i32 %sum.reload3
}
int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N; i++) {
    sum += A[i];
    A[i] = sum;
  }
  return sum;
}

define i32 @f(i32* %A, i32 %N) {
  entry:
    %sum = alloca i32
    store i32* %sum, i32 0
    br label %for.cond

  for.cond:
    %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
    %cmp = icmp slt i32 %iv, %N
    br i1 %cmp, label %for.body, label %for.end

  for.body:
    %idx = getelementptr inbounds i32* %A, i32 %iv
    %tmp1 = load i32* %idx, align 4
    %sum.reload = load i32* %sum
    %add = add nsw i32 %sum.reload, %tmp1
    store i32* %sum, i32 %add
    %sum.reload2 = load i32* %sum
    store i32* %idx, i32 %sum.reload2
    %iv.next = add nuw nsw i32 %iv, 1
    br label %for.cond

  for.end:
    %sum.reload3 = load i32* %sum
    ret i32 %sum.reload3
}
Reduction-like Computation vs. Reduction Dependences

define i32 @f(i32* %A, i32 %N) {
  entry:
    %sum = alloca i32
    store i32* %sum, i32 0
    br label %for.cond
  
  for.cond:
    %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
    %cmp = icmp slt i32 %iv, %N
    br i1 %cmp, label %Stmt.R1, label %for.end
  
  Stmt.R1:
    %sum.reload = load i32* %sum
    %mul = mul nsw i32 %sum.reload, 3
    store i32* %sum, i32 %mul
    br label %Stmt.S
  
  Stmt.S:
    ...
    br label %Stmt.R2
  
  Stmt.R2:
    %idx = getelementptr inbounds i32* %A, i32 %iv
    %tmp1 = load i32* %idx, align 4
    %sum.reload2 = load i32* %sum
    %add = add nsw i32 %sum.reload2, %tmp1
    store i32* %sum, i32 %add
    %iv.next = add nuw nsw i32 %iv, 1
    br label %for.cond
  
  for.end:
    %sum.reload3 = load i32* %sum
    ret i32 %sum.reload3
}

int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N ; i++) {
    R1: sum = sum * 3;
    S(i);
    R2: sum = sum + A[i];
  }
  return sum;
}
Reduction-aware Scheduling by Hand

```c
void f(int *A, long n) {
  for (long i = 0; i < 2 * n; i++)
    S0: A[0] += i;
  for (long i = 0; i < 2 * n; i++)
    S1: A[i + 1] = 1;
}
```

Reduction-aware Scheduling by Hand

```c
void f(int *A, long n) {
    for (long i = 0; i < 2 * n; i++)
    S0: A[0] += i;
    for (long i = 0; i < 2 * n; i++)
    S1: A[i + 1] = 1;
}
```

Schedule:

\[
\begin{align*}
[n] & \rightarrow \{Stmt_{S0}[i0] \rightarrow scattering[0, -i0, 0] : i0\%2 = 0; \\
         Stmt_{S0}[i0] & \rightarrow scattering[2, i0, 0] : i0\%2 = 1; \\
[n] & \rightarrow \{Stmt_{S1}[i0] \rightarrow scattering[1, i0, 0] \}
\end{align*}
\]
Reduction-aware Scheduling by Hand

```c
void f(int *A, long n) {
  for (long i = 0; i < 2 * n; i++)
    S0: A[0] += i;
  for (long i = 0; i < 2 * n; i++)
    S1: A[i + 1] = 1;
}
```

Schedule:

```
[n] → {Stmt_S0[i0] → scattering[0, −i0, 0] : i0%2 = 0;
      Stmt_S0[i0] → scattering[2, i0, 0] : i0%2 = 1};

[n] → {Stmt_S1[i0] → scattering[1, i0, 0]}
```

```c
#pragma known-parallel reduction
for (int c0 = 0; c0 <= 2; c0 += 1) {
  if (c0 == 2) {
    #pragma simd reduction
    for (int c1 = 1; c1 < 2 * n; c1 += 2)
      Stmt_S0(c1);
  } else if (c0 == 1) {
    #pragma simd
    for (int c1 = 0; c1 < 2 * n; c1 += 1)
      Stmt_S1(c1);
  } else
    #pragma simd reduction
    for (int c1 = -2 * n + 2; c1 <= 0; c1 += 2)
      Stmt_S0(-c1);
}
```