Beyond Do Loops: Data Transfer Generation with Convex Array Regions

Serge Guelton, Mehdi Amini, Béatrice Creusillet

Telecom Bretagne, Brest, France
Silkan, Meudon, France
MINES ParisTech / CRI, Fontainebleau, France

LCPC / Tokyo / Japan / September 11–13th 2012
Outline

Motivation

Convex Array Region

Statement Isolation

Communication Optimization

Applications
Problem

Many modern architecture use the load–work–store paradigm
Problem

Many modern architecture use the load–work–store paradigm.

Is it possible to design a generic pass to generate data transfers?
Goals of This Talk

1. Recall Convex Array Regions, a powerful interprocedural analysis
Goals of This Talk

1. Recall Convex Array Regions, a powerful interprocedural analysis

2. Introduce Statement Isolation, a generic code transformation that generates data transfers
Goals of This Talk

1. Recall Convex Array Regions, a powerful interprocedural analysis
2. Introduce Statement Isolation, a generic code transformation that generates data transfers
3. Extend Redundant Load Store Elimination to these data transfers
Goals of This Talk

1. Recall Convex Array Regions, a powerful interprocedural analysis
2. Introduce Statement Isolation, a generic code transformation that generates data transfers
3. Extend Redundant Load Store Elimination to these data transfers
4. Illustrate these transformations on various architectures.
Outline

Motivation

Convex Array Region

Statement Isolation

Communication Optimization

Applications
What is a Convex Array Region?

Convex Array Regions

- Starting with Béatrice CREUSILLET thesis (1996)
- Find out what part of an array is read or written
- Approximation: may/must/exact
- Set of linear relations

Applications

- Parallelization
- Array privatization
- Scalarization
- Statement isolation
- Memory footprint reduction using tiling
Basic Example

```c
//
int triangular(int m, int n, double a[n][m]) {
    int h = n/2;
    //
    //
    for(int i = 0; i < h; i += 1)
        //
        //
        for(int j = i; j < m-i; j += 1)
            //
            //
            a[i][j] = f();
}
```
Basic Example

```
// W(a) = {a[φ1][φ2] | 0 ≤ φ1; φ1 ≤ φ2; φ1 + φ2 + 1 ≤ m; 2 × φ1 + 2 ≤ n}
int triangular(int m, int n, double a[n][m]) {
    int h = n / 2;
    // W(a) = {a[φ1][φ2] | 0 ≤ φ1; φ1 ≤ φ2; φ1 + φ2 + 1 ≤ m;
    // φ1 + 1 ≤ h; 2 × h ≤ n ≤ 2 × h + 1}
    for (int i = 0; i < h; i += 1)
        // W(a) = {a[φ1][φ2] | φ1 == i; i <= φ2; φ2 + i + 1 <= m;
        // 0 <= i ; i + 1 <= h, n <= 2h + 1, 2h <= n}
        for (int j = i; j < m - i; j += 1)
            // W(a) = {a[φ1][φ2] | φ1 == i; φ2 == j; i <= j; j + i + 1 <= m;
            // 0 <= i ; i + 1 <= h, n <= 2h + 1, 2h <= n}
            a[i][j] = f();
}
```
Tricky Examples

Array regions captures function memory accesses for interprocedural propagation

// R(src) = \{src[\phi_1] | i \leq \phi_1 \leq i + k - 1\}
// W(dst) = \{dst[\phi_1] | \phi_1 = i\}
// R(m) = \{m[\phi_1] | 0 \leq \phi_1 \leq k - 1\}
int kernel(int i, int n, int k, int src[n], int dst[n-k],
           int m[k]) {
  int v=0;
  for (int j = 0; j < k; ++j )
    v += src[ i + j ] * m[ j ];
  dst[i]=v;
}

void fir(int n, int k, int src[n], int dst[n-k], int m[k]) {
  for (int i = 0; i < n - k+ 1; ++i )
    // R(src) = \{src[\phi_1] | i \leq \phi_1 \leq i + k - 1, 0 \leq i \leq n - k\}
    // R(m) = \{m[\phi_1] | 0 \leq \phi_1 \leq k - 1\}
    // W(dst) = \{dst[\phi_1] | \phi_1 = i\}
    kernel(i, n, k, src, dst, m);
}
Tricky Examples

Array regions captures function memory accesses for interprocedural propagation

// $R(\text{src}) = \{\text{src}[\phi_1] | i \leq \phi_1 \leq i + k - 1\}$
// $W(\text{dst}) = \{\text{dst}[\phi_1] | \phi_1 = i\}$
// $R(\text{m}) = \{\text{m}[\phi_1] | 0 \leq \phi_1 \leq k - 1\}$

```c
int kernel(int i, int n, int k, int src[n], int dst[n-k],
           int m[k]) {
    int v = 0;
    for (int j = 0; j < k; ++j)
        v += src[i + j] * m[j];
    dst[i] = v;
}
```

```c
void fir(int n, int k, int src[n], int dst[n-k], int m[k]) {
    for (int i = 0; i < n - k + 1; ++i)
        // $R(\text{src}) = \{\text{src}[\phi_1] | i \leq \phi_1 \leq i + k - 1, 0 \leq i \leq n - k\}$
        // $R(\text{m}) = \{\text{m}[\phi_1] | 0 \leq \phi_1 \leq k - 1\}$
        // $W(\text{dst}) = \{\text{dst}[\phi_1] | \phi_1 = i\}$
        kernel(i, n, k, src, dst, m);
}
```
Tricky Examples

Array regions are summarized for a while loop

\[
\bar{R}(\text{randv}) = \{\text{randv}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}
\]
\[
\bar{W}(a) = \{a[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{5N+9}{12}\}
\]

```c
void foo (int N, int a[N], int randv[N]) {
    int x=N/4 ,y =0;
    while (x<=N /3) {
        a[x+y] = x+y;
        if ( randv [x-y]) x = x +2; else x++ ,y ++;
    }
}
```

Array regions captures the accesses for a code with a switch/case

\[
\bar{R}(\text{in}) = \{\text{src}[\phi_1] \mid i \leq \phi_1 \leq i + 2\}
\]
\[
\bar{W}(\text{out}) = \{\text{out}[\phi_1] \mid \phi_1 = i\}
\]

```c
void foo (int n, int i, char c, int out[n], int in[n]) {
    switch (c){
        case 'a': case 'e':
            out[i]=in[i]; break;
        default: out[i]=in[3*(i/3)+2];
    }
}
```
Tricky Examples

Array regions are summarized for a while loop

```c
// \overline{R}(\text{randv}) = \{\text{randv}[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}
// \overline{W}(a) = \{a[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{5N+9}{12}\}

void foo(int N, int a[N], int randv[N]) {
    int x=N/4, y=0;
    while (x <= N /3) {
        a[x+y] = x+y;
        if (randv[x-y]) x = x +2; else x++, y++;
    }
}
```

Array regions captures the accesses for a code with a switch/case

```c
// \overline{R}(\text{in}) = \{\text{src}[\phi_1] | i \leq \phi_1 \leq i+2\}
// \overline{W}(\text{out}) = \{\text{out}[\phi_1] | \phi_1 = i\}

void foo(int n, int i, char c, int out[n], int in[n]) {
    switch (c){
        case 'a': case 'e':
            out[i]=in[i]; break;
        default: out[i]=in[3*(i/3)+2];
    } 
}
```
Outline

Motivation

Convex Array Region

Statement Isolation

Communication Optimization

Applications
Statement Isolation

Description
Transform a piece of code to use new memory locations. Generate data transfers between initial and new memory.

```c
long a = random(); b;
b = 2 * a;
printf("%ld\n", b):
```

```c
long a = random(); b;
long a', b';
a' = a;
b' = 2 * a';
b = b';
printf("%ld\n", b);
```

Objectives

〜 Minimize amount of transferred data.
〜 Transfer parts of arrays.
〜 Use generic data transfer functions
Relationship with Convex Array Regions

Region $\simeq$ DMA

- Write Region $\simeq$ transfer to the host;
- Read Region $\simeq$ transfer from the host.
Relationship with Convex Array Regions

Region $\simeq$ DMA

- Write Region $\simeq$ transfer to the host;
- Read Region $\simeq$ transfer from the host.

Informal Relationship

$\text{Store}(s, \sigma) =$ all written regions
$\text{Load}(s, \sigma) =$ all read regions, + the regions that may not be written
Relationship with Convex Array Regions

Region $\simeq$ DMA

- Write Region $\simeq$ transfer to the host;
- Read Region $\simeq$ transfer from the host.

Informal Relationship

$\text{Store}(s, \sigma) =$ all written regions
$\text{Load}(s, \sigma) =$ all read regions, + the regions that may not be written

Formal Relationship

$$\text{Store}(s, \sigma) = \lceil \overline{W}(s, \sigma) \rceil$$
$$\text{Load}(s, \sigma) = \lceil \overline{R}(s, \sigma) \cup (\text{Store}(s, \sigma) - \overline{W}(s, \sigma)) \rceil$$
Statement Isolation : Example

```c
void erode(int n, int m, int in[n][m], int out[n][m]) {
    // declare isolated variables
    int (*out0)[n][m] = 0, (*in0)[n][m+1] = 0;
    // allocated isolated variables
    accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
    accel_malloc((void **) &out0, sizeof(int)*n*m);
    // transfer data in
    copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0, &in[0][0], *in0);
    copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // execute kernel in isolated memory
    for(int i = 0; i <= n -1; i += 1)
        for(int j = 0; j <= m -1; j += 1)
            if (j==0) (*out0)[i][j] = MIN((*in0)[i][j], (*in0)[i][j+1]);
            else if (j==m -1) (*out0)[i][j] = MIN((*in0)[i][j-1], (*in0)[i][j]);
            else (*out0)[i][j] = MIN((*in0)[i][j-1],(*in0)[i][j],(*in0)[i][j+1]);
    // transfer data out
    copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // free isolated memory
    accel_free(in0);
    accel_free(out0);
}
```
void erode(int n, int m, int in[n][m], int out[n][m]) {
    // declare isolated variables
    int (*out0)[n][m] = 0, (*in0)[n][m+1] = 0;
    // allocated isolated variables
    accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
    accel_malloc((void **) &out0, sizeof(int)*n*m);
    // transfer data in
    copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0, &in[0][0], *in0);
    copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // execute kernel in isolated memory
    for (int i = 0; i <= n -1; i += 1)
        for (int j = 0; j <= m -1; j += 1)
            if (j==0) (*out0)[i][j] = MIN((*in0)[i][j], (*in0)[i][j +1]);
            else if (j==m -1) (*out0)[i][j] = MIN((*in0)[i][j-1], (*in0)[i][j]);
            else (*out0)[i][j] = MIN((*in0)[i][j-1],(*in0)[i][j],(*in0)[i][j +1]);
    // transfer data out
    copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // free isolated memory
    accel_free(in0);
    accel_free(out0);
}
Statement Isolation: Example

```c
void erode ( int n, int m, int in[n][m], int out[n][m]) {
    // declare isolated variables
    int (* out0)[n][m] = 0, (* in0)[n][m+1] = 0;
    // allocated isolated variables
    accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
    accel_malloc((void **) &out0, sizeof(int)*n*m);
    // transfer data in
    copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0, &in[0][0], *in0);
    copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // execute kernel in isolated memory
    for (int i = 0; i <= n -1; i += 1)
        for (int j = 0; j <= m -1; j += 1)
            if (j == 0) (*out0)[i][j] = MIN ((*in0)[i][j], (*in0)[i][j +1]);
            else if (j == m -1) (*out0)[i][j] = MIN ((*in0)[i][j -1], (*in0)[i][j]);
            else (*out0)[i][j] = MIN ((*in0)[i][j -1], (*in0)[i][j], (*in0)[i][j +1]);
    // transfer data out
    copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // free isolated memory
    accel_free(in0);
    accel_free(out0);
}
```
Statement Isolation: Example

```c
void erode(int n, int m, int in[n][m], int out[n][m]) {
    // declare isolated variables
    int (*out0)[n][m] = 0, (*in0)[n][m+1] = 0;
    // allocated isolated variables
    accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
    accel_malloc((void **) &out0, sizeof(int)*n*m);
    // transfer data in
    copy_to_accel_2d(sizeof(int), n, m, n, m+1, 0, 0, &in[0][0], *in0);
    copy_to_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // execute kernel in isolated memory
    for (int i = 0; i <= n-1; i += 1)
        for (int j = 0; j <= m-1; j += 1)
            if (j==0) (*out0)[i][j] = MIN((*in0)[i][j], (*in0)[i][j+1]);
    copy_from_accel_2d(sizeof(int), n, m, n, m, 0, 0, &out[0][0], *out0);
    // free isolated memory
    accel_free(in0);
    accel_free(out0);
}
```
Outline

Motivation

Convex Array Region

Statement Isolation

Communication Optimization

Applications
Redundant Load Store Elimination

Description
Move data transfers upward in the AST to eliminate redundant and/or invariant transfers.

Extension from scalar to any copy function

Constraints

〜 Establish clear definitions of “load” and “store”
〜 Must work for SSE/AVX, FPGA boards or GPU data transfers
〜 Inter-procedural transformation
Redundant Vector Transfer Elimination: Example

Before Optimization

```c
void a(int i, int A[2], int B[2]) {
    while (i-- >= 0) {
        load(B, A);
        B[0]++;
        store(A, B);
    }
}
int main() {
    a(0, A, B);
    a(1, A, B);
    printf("%d\n", A[1]);
    return 0;
}
```

After Optimization

```c
void a(int i, int A[2], int B[2]) {
    while (i-- >= 0) {
        B[0]++;
    }
}
int main() {
    int j[2] = {1, 2}, k[2];
    load(B, A);
    a(0, A, B);
    a(1, A, B);
    store(A, B);
    printf("%d\n", A[1]);
    return 0;
}
```
Redundant Vector Transfer Elimination: Example

Before Optimization

```c
void a(int i, int A[2], int B[2]) {
    while (i-- >= 0) {
        load(B, A);
        B[0]++;
        store(A, B);
    }
}

int main() {
    a(0, A, B);
    a(1, A, B);
    printf("%d\n", A[1]);
    return 0;
}
```

After Optimization

```c
void a(int i, int A[2], int B[2]) {
    while (i-- >= 0) {
        B[0]++;
    }
}

int main() {
    int j[2] = {1, 2}, k[2];
    load(B, A);
    a(0, A, B);
    a(1, A, B);
    store(A, B);
    printf("%d\n", A[1]);
    return 0;
}
```
Redundant Load Store Elimination: Inter-Iterations

// \( \overline{R}(x) = \{x[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 \times \phi_1 + 4; \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n \} \)

// \( \overline{W}(x) = \{x[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 \times \phi_1 + 2; \phi_1 + 2 \leq n \} \)

for (i1 = 0; i1 < n / 2; i1++) { // Sequential

// \( \overline{R}(x) = \{x[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 \times \phi_1 + 4; \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n \} \)

// \( \overline{W}(x) = \{x[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 \times \phi_1 + 2; \phi_1 + 2 \leq n \} \)

for (i2 = i1; i2 < n - i1; i2++) { // Parallel, on an accelerator

// \( \overline{R}(x) = \{x[\phi_1][\phi_2] \mid \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \)

// \( \overline{W}(x) = \{x[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \)

X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
}
}
Redundant Load Store Elimination: Inter-Iterations

\[
\text{// } \bar{R}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 \cdot \phi_1 + 4; \\
\quad \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n \} \\
\text{// } \overline{W}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 \cdot \phi_1 + 2; \phi_1 + 2 \leq n \}
\]

\[
\text{for } (i1=0; i1<n/2; i1++) \{ \text{// Sequential} \\
\text{// } \bar{R}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \\
\text{// } \overline{W}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : \phi_1 + i1 = n - 2; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \\
\}
\]

\[
\text{// } \bar{R}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; \\
\quad i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \\
\text{// } \overline{W}(\mathbf{x}) = \{ \mathbf{x}[\phi_1][\phi_2] : \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \leq i1; i1 \leq i2 \} \\
\mathbf{x}[n - 2 - i1][i2] = \mathbf{x}[n - 2 - i1][i2] - \mathbf{x}[n - i1 - 3][i2]; \\
\}
\]

Read
Read and Written
Written on previous iterations
Redundant Load Store Elimination: Inter-Iterations

// $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 \times \phi_1 + 4; 
\phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n\}$

// $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 \times \phi_1 + 2; \phi_1 + 2 \leq n\}$

for (i1 = 0; i1 < n / 2; i1++) { // Sequential

// $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$

// $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$

for (i2 = i1; i2 < n - i1; i2++) { // Parallel, on an accelerator

// $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; 
i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$

// $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \leq i1; i1 \leq i2\}$

$X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];$
}

}
Redundant Load Store Elimination: Inter-Iterations

// $\bar{R}(x) = \{x[\phi_1][\phi_2] | \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 \cdot \phi_1 + 4; \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n\}$

// $\bar{W}(x) = \{x[\phi_1][\phi_2] | \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 \cdot \phi_1 + 2; \phi_1 + 2 \leq n\}$

for (i1 = 0; i1 < n/2; i1++) { // Sequential
  // $\bar{R}(x) = \{x[\phi_1][\phi_2] | n \leq \phi_1 + i_1 + 3; \phi_1 + i_1 + 2 \leq n; i_1 \leq \phi_2; \phi_2 + i_1 + 1 \leq n\}$
  // $\bar{W}(x) = \{x[\phi_1][\phi_2] | \phi_1 + i_1 = n - 2; i_1 \leq \phi_2; \phi_2 + i_1 + 1 \leq n\}$

  // $\bar{R}(x) = \{x[\phi_1][\phi_2] | \phi_2 = i_2; n \leq \phi_1 + i_1 + 3; \phi_1 + i_1 + 2 \leq n; i_1 \leq \phi_2; \phi_2 + i_1 + 1 \leq n\}$
  // $\bar{W}(x) = \{x[\phi_1][\phi_2] | \phi_1 + i_1 = n - 2; \phi_2 = i_2; 0 \leq i_1; i_1 \leq i_2\}$
  X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
}

X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
Redundant Load Store Elimination: Inter-Iterations

// $R(X) = \{X[\phi_1][\phi_2] | \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 * \phi_1 + 4; \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n\}$

// $W(X) = \{X[\phi_1][\phi_2] | \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 * \phi_1 + 2; \phi_1 + 2 \leq n\}$

for (i1 = 0; i1 < n/2; i1++) { // Sequential
    // $R(X) = \{X[\phi_1][\phi_2] | n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
    // $W(X) = \{X[\phi_1][\phi_2] | \phi_1 + i1 = n - 2; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$

    // $R(X) = \{X[\phi_1][\phi_2] | \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
    // $W(X) = \{X[\phi_1][\phi_2] | \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \leq i1; i1 \leq i2\}$
    X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
}

X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];

Read
Read and Written
Written on previous iterations
for (i1 = 0; i1 < n/2; i1++) { // Sequential
    // Allocate all the array on the accelerator
    double (*accel_X)[2][-2*i1+n];
    P4A_accel_malloc((void **) &accel_X, sizeof(double)*i1*2));
    Copy_to_accel_2d(sizeof(double), n, n, 2, -2*i1+n, -i1+n-3, i1, &X[0][0], *
        accel_X);
    for (i2 = 0; i2 < n-i1-i1; i2++) { // Parallel (has been skewed to start from 0)
        accel_X[1][i2] = accel_X[1][i2] - accel_X[0][i2];
    }
    Copy_from_accel_2d(
        sizeof(double),
        n, n, // host size
        1, -2*i1+n, // transfer
        -i1+n-2, i1, // offset
        &X[0][0],
        &accel_X[1][0]);
    Accel_free(accel_X);
}
Redundant Load Store Elimination: Inter-Iterations

```c
for (i1 = 0; i1 < n/2; i1++) {  // Sequential
    // Allocate all the array on the accelerator
    double (*accel_X)[2][-2*i1+n];
    P4A_accel_malloc((void **) &accel_X, sizeof(double)*i1*2);
    Copy_to_accel_2d(sizeof(double), n, n, 2, -2*i1+n, -i1+n-3, i1, &X[0][0], *accel_X);
    for (i2 = 0; i2 < n-i1-i1; i2++) {  // Parallel (has been skewed to start from 0)
        accel_X[1][i2] = accel_X[1][i2] - accel_X[0][i2];
    }
    Copy_from_accel_2d(
        sizeof(double),
        n, n, // host size
        1, -2*i1+n, // transfer
        -i1+n-2, i1, // offset
        &X[0][0],
        &accel_X[1][0]);
    Accel_free(accel_X);
}
```
Avoid Redundant Transfers? Try Regions Subtraction...

// \( R(X) = \{ X[\phi_1][\phi_2] \mid n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \} \)

\[ 
\begin{align*}
// R(X) &= \{ X[\phi_1][\phi_2] \mid n \leq \phi_1 + (i1 - 1) + 3; \phi_1 + (i1 - 1) + 2 \leq n; (i1 - 1) \leq \phi_2; \phi_2 + (i1 - 1) + 1 \leq n \} \\
// R(X) &= \{ X[\phi_1][\phi_2] \mid n = \phi_1 + i1 + 3; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n \}
\end{align*}
\]
double (*accel_X)[n-2-(n/2-1)+1][n-1+1];
accel_malloc((void **) &accel_X, sizeof(double)*(n-2-(n/2-1)
+1)*(n-1+1));
// Data for first iteration
copy_to_accel_2d(sizeof(double), n, n, 1, n, n-3, 0, &X
[0][0], &accel_X[n-2-(n/2-1)+1][0]);
for(i1 = 0; i1 < n/2; i1++) { // Sequential
    copy_to_accel_2d(sizeof(double), n, n, 1,-2*i1+n,-i1+n
    -3-2-(n/2-1)+1, i1, &X[0][0],*accel_X);
    for(i2 = 0; i2 < n-i1-i1; i2++) // Parallel
        X[n-2 -i1-2-(n/2-1)+1][i2] = X[n-2-i1-2-(n/2-1)+1][i2]-
        X[n-i1-3-2-(n/2-1)+1][i2];
    copy_from_accel_2d(n
    sizeof(double),
    n, n, // host size
    1, -2*i1+n, // transfer
    -i1+n-2, i1, // offset
    &X[0][0],
    &accel_X[1][0]);
}
accel_free(accel_X);
Outline

Motivation

Convex Array Region

Statement Isolation

Communication Optimization

Applications
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.
Applications

Array regions were successfully used in compilers for various targets

- Vector registers load/store,
- communication generation for an image-processing dedicated accelerator based on FPGA,
- inter-tasks communications generation for an asymmetric MPSoC,
- specific accelerators with codes involving data transfers between different fields of a structure,
- GPU communication generation in the context of Par4All.

The key point is to abstract data transfers and manipulate them.
Conclusion

- **Convex array regions** are an interesting compromise between accuracy and performance and are applicable to a wide-range of programs,
- **statement isolation** takes advantage of the parallel between convex array regions and data transfers,
- **redundant load store elimination** can be used to further optimize data movements,
- these passes are relatively **independent** from their target,
- they have been successfully used on a wide range of applications.