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

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

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

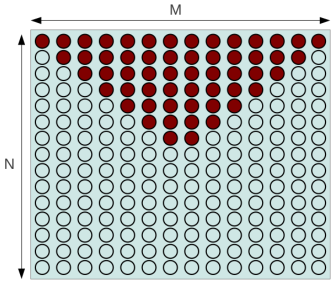
- Parallelization
- Array privatization
- Scalarization
- Statement isolation
- Memory footprint reduction using tiling

Basic Example

```
//
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

```
// $\overline{\mathcal{W}}(\mathtt{a}) = \{\mathtt{a}[\phi_1][\phi_2] \mid 0 \le \phi_1; \phi_1 \le \phi_2; \phi_1 + \phi_2 + 1 \le m; 2 \times \phi_1 + 2 \le n\}$
int triangular(int m, int n, double a[n][m]) {
  int h = n/2;
  // $\mathcal{W}(\mathtt{a}) = \{\mathtt{a}[\phi_1][\phi_2] \mid 0 \le \phi_1; \phi_1 \le \phi_2; \phi_1 + \phi_2 + 1 \le m;$
  // $\phi_1 + 1 \le h; 2 \times h \le n \le 2 \times h + 1\}$
  for(int i = 0; i < h; i += 1)
    // $\mathcal{W}(\mathtt{a}) = \{\mathtt{a}[\phi_1][\phi_2] \mid \phi_1 == i; i <= \phi_2; \phi_2 + i + 1 <= m;$
    // 0<=i ; $i + 1 <= h, n <= 2h + 1, 2h <= n\}$
    for(int j = i; j < m-i; j += 1)
      // $\mathcal{W}(\mathtt{a}) = \{\mathtt{a}[\phi_1][\phi_2] \mid \phi_1 == i; \phi_2 == j; i \le j; j + i + 1 <= m;$
      // 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

```
// $\mathcal{R}(\texttt{src}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1\}$
// $\mathcal{W}(\texttt{dst}) = \{\texttt{dst}[\phi_1] \mid \phi_1 = i\}$
// $\mathcal{R}(\texttt{m}) = \{\texttt{m}[\phi_1] \mid 0 \leq \phi_1 \leq k - 1\}$
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 )
    // $\mathcal{R}(\texttt{src}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1, 0 \leq i \leq n - k\}$
    // $\mathcal{R}(\texttt{m}) = \{\texttt{m}[\phi_1] \mid 0 \leq \phi_1 \leq k - 1\}$
    // $\mathcal{W}(\texttt{dst}) = \{\texttt{dst}[\phi_1] \mid \phi_1 = i\}$
    kernel(i, n, k, src, dst, m);
}
```

Tricky Examples

Array regions captures function memory accesses for interprocedural propagation

```
// $\mathcal{R}(\texttt{src}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1\}$
// $\mathcal{W}(\texttt{dst}) = \{\texttt{dst}[\phi_1] \mid \phi_1 = i\}$
// $\mathcal{R}(\texttt{m}) = \{\texttt{m}[\phi_1] \mid 0 \leq \phi_1 \leq k - 1\}$
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 )
    // $\mathcal{R}(\texttt{src}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1, 0 \leq i \leq n - k\}$
    // $\mathcal{R}(\texttt{m}) = \{\texttt{m}[\phi_1] \mid 0 \leq \phi_1 \leq k - 1\}$
    // $\mathcal{W}(\texttt{dst}) = \{\texttt{dst}[\phi_1] \mid \phi_1 = i\}$
    kernel(i, n, k, src, dst, m);
}
```

Tricky Examples

Array regions are summarized for a while loop

```
// $\overline{\mathcal{R}}(\texttt{randv}) = \{\texttt{randv}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}$
// $\overline{\mathcal{W}}(\texttt{a}) = \{\texttt{a}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{5*N+9}{12}\}$
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

```
// $\overline{\mathcal{R}}(\text{in}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i+2\}$
// $\overline{\mathcal{W}}(\text{out}) = \{\texttt{out}[\phi_1] \mid \phi_1 = i\}$
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];
  }
}
```

Tricky Examples

Array regions are summarized for a while loop

```
// $\overline{\mathcal{R}}(\texttt{randv}) = \{\texttt{randv}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}$
// $\overline{\mathcal{W}}(\texttt{a}) = \{\texttt{a}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{5*N+9}{12}\}$
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

```
// $\overline{\mathcal{R}}(\texttt{in}) = \{\texttt{src}[\phi_1] \mid i \leq \phi_1 \leq i+2\}$
// $\overline{\mathcal{W}}(\texttt{out}) = \{\texttt{out}[\phi_1] \mid \phi_1 = i\}$
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

Statement Isolation

Description

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

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

$\rightsquigarrow$

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

Objectives

- $\rightsquigarrow$ Minimize amount of transfered data.
- $\rightsquigarrow$ Transfer parts of arrays.
- $\rightsquigarrow$ Use generic data transfer functions

Relationship with Convex Array Regions

Region ≃ DMA

- Write Region ≃ transfer to the host ;
- Read Region ≃ 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

$Store(s, \sigma)$ =all written regions

$Load(s, \sigma)$ =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

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

Formal Relationship

$$\begin{aligned} Store(s,\sigma) =& \lceil \overline{\mathcal{W}}(s,\sigma) \rceil \\ Load(s,\sigma) =& \lceil \overline{\mathcal{R}}(s,\sigma) \cup (Store(s,\sigma) - \underline{\mathcal{W}}(s,\sigma)) \rceil \end{aligned}$$

Statement Isolation : Example

```
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

```
int (*out0)[n][m] = 0, (*in0)[n][m+1] = 0;
```

```
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

```
accel_malloc((void **) &in0, sizeof(int)*n*(m+1));
accel_malloc((void **) &out0, sizeof(int)*n*m);
```

```
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);
```

```
accel_free(in0);
accel_free(out0);
```

Statement Isolation : Example

```
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
```

```
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

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

```
void a(int i, int A[2], int B
    [2]) {
  while (i-->=0) {
    load(B, A);
    B[0]++;
    store(A, B);
  }
}
int main() {
  int A[2] = {1, 2}, B[2];

  a(0, A, B);
  a(1, A, B);

  printf("%d\n", A[1]);
  return 0;
}
```

After Optimization

```
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

```
void a(int i, int A[2], int B
    [2]) {
  while (i-->=0) {
    load(B, A);
    B[0]++;
    store(A, B);
  }
}
int main() {
  int A[2] = {1, 2}, B[2];

  a(0, A, B);
  a(1, A, B);

  printf("%d\n", A[1]);
  return 0;
}
```

After Optimization

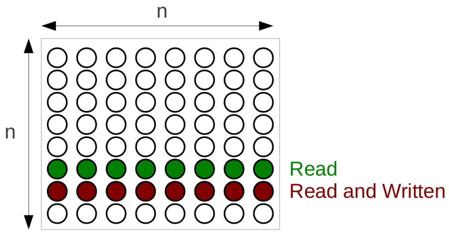
```
void a(int i, int A[2], int B
    [2]) {
  while (i-->=0) {

    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

```
// $\overline{\mathcal{R}}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 * \phi_1 + 4; \phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n\}$
// $\overline{\mathcal{W}}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 * \phi_1 + 2; \phi_1 + 2 \leq n\}$
for(i1=0;i1<n/2;i1++){ // Sequential
 // $\mathcal{R}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
 // $\mathcal{W}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
 for(i2=i1;i2<n-i1;i2++){ // Parallel, on an accelerator
  // $\mathcal{R}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
  // $\mathcal{W}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \leq i1; i1 \leq i2\}$
  X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
 }
}
```

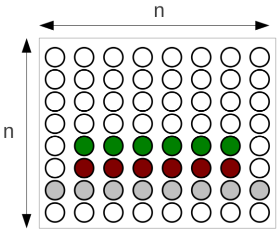


Redundant Load Store Elimination : Inter-Iterations

```
// $\overline{\mathcal{R}}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 2; n \leq \phi_1 + \phi_2 + 3; n \leq 2 * \phi_1 + 4;$
                         $\phi_1 + 2 \leq n; 0 \leq \phi_2; \phi_2 + 1 \leq n; 2 \leq n\}$
// $\overline{\mathcal{W}}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 \leq \phi_1 + 1; n \leq \phi_1 + \phi_2 + 2; n \leq 2 * \phi_1 + 2; \phi_1 + 2 \leq n\}$
for(i1=0;i1<n/2;i1++){ // Sequential
                                                                    $\leq n\}$
  for(i2=i1;i2<n-i1;i2++){ // Parallel, on an accelerator
    // $\mathcal{R}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_2 = i2; n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n;$
                         $i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$
    // $\mathcal{W}(\texttt{X}) = \{\texttt{X}[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \leq i1; i1 \leq i2\}$
    X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
  }
}
```

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

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

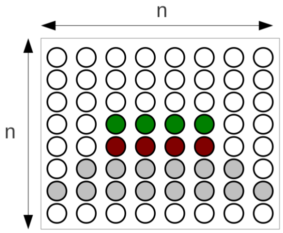


Redundant Load Store Elimination : Inter-Iterations

```
// $\overline{\mathcal{R}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 2; n \le \phi_1 + \phi_2 + 3; n \le 2 * \phi_1 + 4;$
                              $\phi_1 + 2 \le n; 0 \le \phi_2; \phi_2 + 1 \le n; 2 \le n\}$
// $\overline{\mathcal{W}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 1; n \le \phi_1 + \phi_2 + 2; n \le 2 * \phi_1 + 2; \phi_1 + 2 \le n\}$
for(i1=0;i1<n/2;i1++){ // Sequential
                                                                          $\le n\}$
  for(i2=i1;i2<n-i1;i2++){ // Parallel, on an accelerator
    // $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 = i2; n \le \phi_1 + i1 + 3; \phi_1 + i1 + 2 \le n;$
                              $i1 \le \phi_2; \phi_2 + i1 + 1 \le n\}$
    // $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \le i1; i1 \le i2\}$
    X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
  }
}
```

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

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



Read
Read and Written
Written on previous iterations

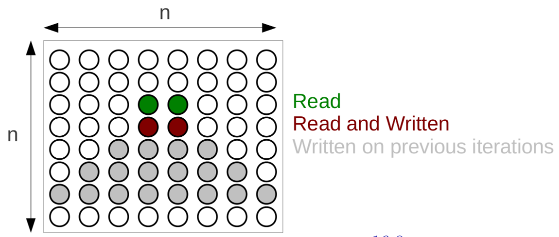
Redundant Load Store Elimination : Inter-Iterations

```
// $\overline{\mathcal{R}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 2; n \le \phi_1 + \phi_2 + 3; n \le 2 * \phi_1 + 4;$
                              $\phi_1 + 2 \le n; 0 \le \phi_2; \phi_2 + 1 \le n; 2 \le n\}$
// $\overline{\mathcal{W}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 1; n \le \phi_1 + \phi_2 + 2; n \le 2 * \phi_1 + 2; \phi_1 + 2 \le n\}$
for(i1=0;i1<n/2;i1++){ // Sequential
                                                                        $\le n\}$
```

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

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

```
 for(i2=i1;i2<n-i1;i2++){ // Parallel, on an accelerator
  // $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 = i2; n \le \phi_1 + i1 + 3; \phi_1 + i1 + 2 \le n;$
                              $i1 \le \phi_2; \phi_2 + i1 + 1 \le n\}$
  // $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \le i1; i1 \le i2\}$
  X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
 }
}
```

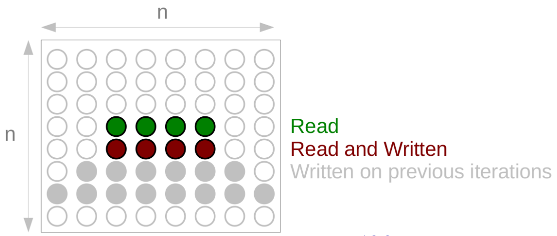


Redundant Load Store Elimination : Inter-Iterations

```
// $\overline{\mathcal{R}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 2; n \le \phi_1 + \phi_2 + 3; n \le 2 * \phi_1 + 4;$
                   $\phi_1 + 2 \le n; 0 \le \phi_2; \phi_2 + 1 \le n; 2 \le n\}$
// $\overline{\mathcal{W}}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 \le \phi_1 + 1; n \le \phi_1 + \phi_2 + 2; n \le 2 * \phi_1 + 2; \phi_1 + 2 \le n\}$
for(i1=0;i1<n/2;i1++){ // Sequential
```

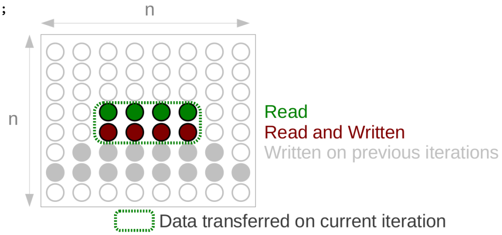
// $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid n \le \phi_1 + i1 + 3; \phi_1 + i1 + 2 \le n; i1 \le \phi_2; \phi_2 + i1 + 1 \le n\}$
// $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; i1 \le \phi_2; \phi_2 + i1 + 1 \le n\}$

```
                                                                          $\le n\}$
  for(i2=i1;i2<n-i1;i2++){ // Parallel, on an accelerator
    // $\mathcal{R}(X) = \{X[\phi_1][\phi_2] \mid \phi_2 = i2; n \le \phi_1 + i1 + 3; \phi_1 + i1 + 2 \le n;$
                   $i1 \le \phi_2; \phi_2 + i1 + 1 \le n\}$
    // $\mathcal{W}(X) = \{X[\phi_1][\phi_2] \mid \phi_1 + i1 = n - 2; \phi_2 = i2; 0 \le i1; i1 \le i2\}$
    X[n - 2 - i1][i2] = X[n - 2 - i1][i2] - X[n - i1 - 3][i2];
  }
}
```



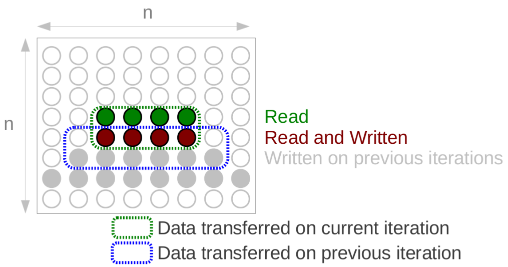
Redundant Load Store Elimination : Inter-Iterations

```
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);
}
```


n
n
Read
Read and Written
Written on previous iterations
Data transferred on current iteration

Redundant Load Store Elimination : Inter-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);
}
```


n
n
Read
Read and Written
Written on previous iterations
Data transferred on current iteration
Data transferred on previous iteration

Avoid Redundant Transfers ? Try Regions Subtraction...

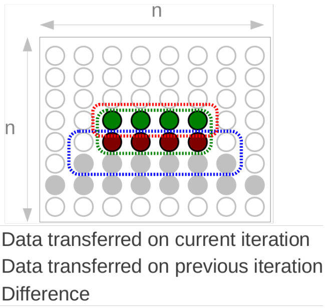
$$// \mathcal{R}(\mathrm{X}) = \{\mathrm{X}[\phi_1][\phi_2] \mid n \leq \phi_1 + i1 + 3; \phi_1 + i1 + 2 \leq n; i1 \leq \phi_2; \phi_2 + i1 + 1 \leq n\}$$

$$-$$

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

$$=$$

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


n
n
Data transferred on current iteration
Data transferred on previous iteration
Difference

Pipelined Overlapped Communications and Computations

```
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

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.