Beyond Do Loops: Data Transfer Generation with Convex Array Regions

Serge Guelton[1], Mehdi Amini[2,3], Béatrice Creusillet[3]

[1] Telecom Bretagne, Brest, France, name.surname@telecom-bretagne.fr
[2] MINES ParisTech/CRI, Fontainebleau, France, name.surname@mines-paristech.fr
[3] HPC-Project, Meudon, France, name.surname@hpc-project.com

Abstract. Automatic data transfer generation is a critical step for guided or automatic code generation for accelerators using distributed memories. Although good results have been achieved for loop nests, more complex control flows such as switches or while loops are generally not handled. This paper shows how to leverage the convex array regions abstraction to generate data transfers. The scope of this study ranges from inter-procedural analysis in simple loop nests with function calls, to inter-iteration data reuse optimization and arbitrary control flow in loop bodies. Generated transfers are approximated when an exact solution cannot be found. Array regions are also used to extend redundant load store elimination to array variables. The approach has been successfully applied to GPUs and domain-specific hardware accelerators.

Keywords: data transfers, convex array regions, redundant transfer elimination, GPU

1 Introduction

The last decade has been showcased by the frequency wall limitation and the beginning of a computing era based on parallel computing. One of the solutions that emerges is based on the use of hardware accelerators, for instance Graphical Processing Units (GPUs). These are massively parallel pieces of hardware, usually plugged in a host computer using the PCI-Express bus, that can provide important performance improvements for data-parallel program.

The main drawback of these accelerators lies in their programming model. There are two major points: first the programmer has to exhibit in some way the huge amount of parallelism required to fulfill the accelerator capacity; second, since the accelerator is plugged in the system and embeds its own memory, the programmer has to explicitly manage Direct Memory Access (DMA) transfers between the main host memory and the accelerator memory.

The first point has been addressed in different ways using dedicated languages/libraries like Thrust [1], with directives over plain C or Fortran [13,26,19], or through automatic code parallelization [5,6,25]. The second point has been

[1] http://thrust.github.com/

addressed using simplified input from the programmer [13,27,19], or automatically [4,24,1,26] using compilers.

This paper exposes how the array regions abstraction [11] can be used by a compiler to automatically compute memory transfers in presence of complex code patterns. Three examples are used throughout the paper to illustrate the approach: Listing 1.1 requires interprocedural array accesses analysis, and Listing 1.2 contains a while loop, for which the memory access pattern requires an approximated analysis.

This paper is organized as follows: array region analyses are first presented in Section 2; then Section 3 introduces the basis of *statement isolation*, a compiler pass that transforms a statement into a statement executed in a separate memory space. A redundant transfer elimination algorithm based on array regions is then introduced in Section 4 to optimize the generated data transfers. Finally, some applications are detailed in Section 5.

```
1  // R(src) = {src[φ1] | i ≤ φ1 ≤ i + k − 1}
   // W(dst) = {dst[φ1] | φ1 = i}
3  // R(m) = {m[φ1] | 0 ≤ φ1 ≤ k − 1}
   int kernel(int i, int n, int k, int src[n], int dst[n-k], int
        m[k]) {
5    int v=0;
     for( int j = 0; j < k; ++j )
7      v += src[ i + j ] * m[ j ];
     dst[i]=v;
9  }
   void fir( int n, int k, int src[n], int dst[n-k], int m[k]) {
11   for( int i = 0; i < n - k+ 1; ++i )
       // R(src) = {src[φ1] | i ≤ φ1 ≤ i + k − 1, 0 ≤ i ≤ n − k}
13     // R(m) = {m[φ1] | 0 ≤ φ1 ≤ k − 1}
       // W(dst) = {dst[φ1] | φ1 = i}
15     kernel(i, n, k, src, dst, m);
   }
```

Listing 1.1: Array regions on a code with a function call.

```
   // R̄(randv) = {randv[φ1] | (N−3)/4 ≤ φ1 ≤ N/3}
2  // W̄(a) = {a[φ1] | (N−3)/4 ≤ φ1 ≤ (5∗N+9)/12}
   void foo(int N, int a[N], int randv[N]) {
4    int x=N/4,y=0;
     while(x<=N/3) {
6      a[x+y] = x+y;
       if (randv[x-y]) x = x+2; else x++,y++;
8    }
   }
```

Listing 1.2: Array regions on a code with a while loop.

2 Introducing Convex Array Regions

Convex array regions were first introduced by Triolet [23] with the initial purpose of summarizing the memory accesses performed on array element sets by function calls. The concept was later generalized and formally defined for any program statement by Creusillet [11,12] and implemented in the Paralléliseur Interprocedural de Programmes Scientifiques (PIPS) compiler framework.

Informally, the READ (resp. WRITE) regions for a statement s are the set of all scalar variables and array elements that are read (resp. written) during the execution of s. This set generally depends on the values of some program variables at the entry point of statement s: the READ regions are said to be a function of the memory store σ preceding the statement execution, and they are collectively denoted $\mathcal{R}(s,\sigma)$ (resp. $\mathcal{W}(s,\sigma)$ for the WRITE regions).

For instance the READ regions for the statement on line 6 in Figure 1.1 are these:

$$\mathcal{R}(s,\sigma) = \{\{\mathtt{v}\},\{\mathtt{i}\},\{\mathtt{j}\},\{\mathtt{src}(\phi_1) \mid \phi_1 = \sigma(\mathtt{i}) + \sigma(\mathtt{j})\},\{\mathtt{m}(\phi_1) \mid \phi_1 = \sigma(\mathtt{j})\}\}$$

where ϕ_x is used to describe the constraints on the xth dimension of an array, and where $\sigma(\mathtt{i})$ denotes the value of the program variable `i` in the memory store σ. From this point, i is used instead of $\sigma(\mathtt{i})$ when there is no ambiguity.

The regions given above correspond to a very simple statement; however, they can be computed for every level of compound statements. For instance, the READ regions of the `for` loop on line 6 in the code in Figure 1.1 are these:

$$\mathcal{R}(s,\sigma) = \{\{\mathtt{v}\},\{\mathtt{i}\},\{\mathtt{src}(\phi_1) \mid i \leq \phi_1 \leq i + k - 1\},\{\mathtt{m}(\phi_1) \mid 0 \leq \phi_1 \leq k - 1\}\}$$

However, computing exact sets is not always possible, either because the compiler lacks information about the values of variables or the program control flow, or because the regions cannot be exactly represented by a convex polyhedron. In these cases, over-approximated convex sets (denoted $\overline{\mathcal{R}}$ and $\overline{\mathcal{W}}$) are computed. In the following example, the approximation is due to the fact that the exact set contains holes, and cannot be represented by a convex polyhedron:

$$\overline{\mathcal{W}}([\![\mathtt{for(int\ i=0;\ i<n;\ i++)if\ (i\ !=\ 3)a[i]=0;}]\!],\sigma) = \{\{\mathtt{n}\},\{\mathtt{a}[\phi_0] \mid 0 \leq \phi_0 < n\}\}$$

whereas in the next example, the approximation is due to the fact that the condition and its negation are nonlinear expressions that cannot be represented exactly in our framework:

$$\overline{\mathcal{R}}([\![\mathtt{if(a[i]>3)b[i]=1;\ else\ c[i]=1}]\!],\sigma) = \{\{\mathtt{i}\},\{\mathtt{a}[\phi_0] \mid \phi_0 = i\},\{\mathtt{b}[\phi_0] \mid \phi_0 = i\},\{\mathtt{c}[\phi_0] \mid \phi_0 = i\}\}$$

Under-approximations (denoted $\underline{\mathcal{R}}$ and $\underline{\mathcal{W}}$) are required when computing region differences (see [10] for more details on approximations when using the convex polyhedron lattice).

READ and WRITE regions summarize the effects of statements and functions upon array elements, but they do not take into account the flow of array element

values. For that purpose, IN and OUT regions have been introduced in [11] to take array kills into account, that is to say, redefinitions of individual array elements:

- IN regions contain the array elements whose values are *imported* by the considered statement, which means the elements that are read before being possibly redefined by another instruction of the statement.
- OUT regions contain the array elements defined by the considered statement, which are used afterwards in the program continuation. They are the *live* or *exported* array elements.

As for READ and WRITE regions, IN and OUT regions may be over- or under-approximated.

There is a strong analogy between the array regions of a statement and the memory used in this statement, at least from an external point of view, which means excluding its privately declared variables. Intuitively, the memory footprint of a statement can be obtained by counting the points in its associated array regions. In the same way, the READ (or IN) and WRITE (or OUT) regions can be used to compute the memory transfers required to execute this statement in a new memory space built from the original space. This analogy is analyzed and leveraged in this paper and especially in Section 3.

3 Communications Code Generation

This section introduces a new generic code transformation, called *statement isolation*. It turns a statement s into a new statement Isol(s) that shares no memory area with the remainder of the code, and is surrounded by the required memory transfers between the two memory spaces. In other words, if s is evaluated in a memory store σ, Isol(s) does not reference any element of σ. The generated memory transfers to and from the new memory space ensure the consistency and validity of the values used in the extended memory space during the execution of Isol(s) and once back to the original execution path.

This transformation assumes no aliasing between the different variables referenced by s, so that array regions of two different variables cannot overlap. It is applicable to any statement for which the array region can be computed, either exactly or approximately.

The transformation is formally described in [15]. To illustrate how the convex array regions are leveraged, the `while` loop in Figure 1.2 is used as an example. The exact and over-approximated array regions for this statement are as follows:

$$\mathcal{R} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{R}}(\mathrm{randv}) = \{\mathrm{randv}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\}$$

$$\mathcal{W} = \{\{\mathrm{x}\}, \{\mathrm{y}\}\} \qquad \overline{\mathcal{W}}(\mathrm{a}) = \{\mathrm{a}[\phi_1] \mid \frac{N-3}{4} \leq \phi_1 \leq \frac{5*N+9}{12}\}$$

The basic idea is to turn each region into a newly allocated variable, large enough to hold the region, then to generate data transfers from the original variables to the new ones, and finally to perform the required copy from the new variables

to the original ones. This results in the code shown in Figure 1.3, where isolated variables have been put in uppercase. Statements `(3)` and `(5)` correspond to the exact regions on scalar variables. Statements `(2)` and `(4)` correspond to the over-approximated regions on array variables. Statement `(1)` is used to ensure data consistency, as explained later.

Notice how `memcpy` system calls are used here to simulate data transfers, and, in particular, how the sizes of the transfers are constrained with respect to the array regions.

```
void foo(int N, int a[N], int randv[N]) {
  int x=0,y=0;
  int A[N/6], RANDV[(N-9)/12], X, Y;
  memcpy(A, a+(N-3)/4, N/6*sizeof(int));                  //(1)
  memcpy(RANDV, randv+(N-3)/4, (N-9)/12*sizeof(int));     //(2)
  memcpy(&X, &x, sizeof(x)); memcpy(&Y, &y, sizeof(y)); //(3)
  while(X<=N/3) {
    A[X+Y-(N-3)/4] = X+Y;
    if (RANDV[X+Y-(N-3)/4]) X = X+2; else X++,Y++;
  }
  memcpy(a+(N-3)/4, A, N/6*sizeof(int));                  //(4)
  memcpy(&x, &X, sizeof(x)); memcpy(&y, &Y, sizeof(y)); //(5)
}
```

Listing 1.3: Isolation of an irregular while loop using array region analysis.

The benefits of using new variables to simulate the extended memory space and of relying on a regular function to simulate the DMA are twofold:

1. The generated code can be executed on a general-purpose processor. It makes it possible to verify and validate the result without the need of an accelerator or a simulator.
2. The generated code is independent of the hardware target: specializing its implementation for a given accelerator requires only a specific implementation of the memory transfer instructions (here `memcpy`).

Converting convex array regions into data transfers From this point on, the availability of data transfer operators that can transfer rectangular subparts of n-dimensional arrays to or from the accelerator is assumed. For instance,

```
size_t memcpy2d(void* dest, void* src,
                size_t dim1, size_t offset1, size_t count1,
                size_t dim2, size_t offset2, size_t count2);
```

copies from `src` to `dest` the rectangular zone between (`offset1`, `offset2`) and (`offset1` + `count1`, `offset2` + `count2`). `dim1` and `dim2` are the sizes of the memory areas pointed to by `src` and `dest` on the host memory, and are used to compute the addresses of the memory elements to transfer.

We show how convex array regions are used to generate calls to these operators. Let src be a n-dimensional variable, and $\{src[\phi_1]\dots[\phi_n] \,|\, \psi(\phi_1,\dots,\phi_n)\}$ be a convex region of this variable.

As native DMA instructions are very seldom capable of transferring anything other than a rectangular memory area, the rectangular hull, denoted $\lceil\cdot\rceil$, is first computed so that the region is expressed in the form

$$\{src[\phi_1]\dots[\phi_n] \,|\, \alpha_1 \leq \phi_1 < \beta_1, \dots, \alpha_n \leq \phi_n < \beta_n\}$$

This transformation can lead to a loss of accuracy and the region approximation can thus shift from *exact* to *may*. This shift is performed when the original region is not equal to its rectangular envelope.

The call to the transfer function can then be generated with $\texttt{offset}k = \alpha_k$ and $\texttt{count}k = \beta_n - \alpha_k$ for each k in $[1\dots n]$.

For a statement s, the memory transfers from σ are generated using its read regions ($\mathcal{R}(s,\sigma)$): any array element read by s must have an up-to-date value in the extended memory space with respect to σ. Symmetrically, the memory transfers back to σ must include all updated values, represented by the written regions ($\mathcal{W}(s,\sigma')$), where σ' is the memory state once s is executed from σ. [2]

However, if the written region is over-approximated, part of the values it contains may not have been updated by the execution of Isol(s). Therefore, to guarantee the consistency of the values transferred *back* to σ, they must first be correctly initialized during the transfer *from* σ. These observations lead to the following equations for the convex array regions transferred from and to σ, respectively denoted $Load(s,\sigma)$ and $Store(s,\sigma)$:

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

$Load(s,\sigma)$ and $Store(s,\sigma)$ are rectangular regions by definition and can be converted into memory transfers, as detailed previously. The new variables with *ad-hoc* dimensions are declared and a substitution taking into account the shifts is performed on s to generate Isol(s).

Managing Variable Substitutions For each variable `v` to be transferred according to $Load(s,\sigma)$, a new variable `V` is declared, which must contain enough space to hold the loaded region. For instance if `v` holds short integers and

$$Load(s,\sigma) = \{\texttt{v}[\phi_1][\phi_2] \,|\, \alpha_1 \leq \phi_1 < \beta_1, \alpha_2 \leq \phi_2 < \beta_2\}$$

then V will be declared as `short int` $\texttt{V}[\beta_1 - \alpha_1][\beta_2 - \alpha_2]$. The translation of an intraprocedural reference to `v` into a reference to `V` is straightforward as $\forall i,j,\ \texttt{V}[i][j] = \texttt{v}[i+\alpha_1][j+\alpha_2]$.

[2] Most of the time, variables used in the region description are not modified by the isolated statement and we can safely use $\mathcal{W}(s,\sigma)$. Otherwise, e.g. `a[i++]=1`, methods detailed in [11] must be applied to express the region in the right memory store.

The combination of this variable substitution with convex array regions is what makes the isolate statement a powerful tool: all the complexity is hidden by the region abstraction.

For interprocedural translation, a new version of the called function is created using the following scheme: for each transferred variable passed as an actual parameter, and for each of its dimensions, an extra parameter is added to the call and to the new function, holding the value of the corresponding offset. These extra parameters are then used to perform the translation in the called function.

The output of the whole process applied to the outermost loop of the Finite Impulse Response (FIR) is illustrated in Figure 1.4, where a new `KERNEL` function with two extra parameters is now called instead of the original `kernel` function. These parameters hold the offsets between the original array variables `src` and `m` and the isolated ones `SRC` and `M`.

```
void fir( int n, int k, int src[n], int dst[n-k], int m[k]) {
  int N=n - k+ 1;
  for( int i = 0; i < N; ++i ) {
    int DST[1],SRC[k],M[k];
    memcpy(SRC, src+i, k*sizeof(int));
    memcpy(M, m+0, k*sizeof(int));
    KERNEL(i, n, k, SRC, DST, M, i/*SRC*/, i/*DST*/, 0/*M*/);
    memcpy(dst, DST+0, 1*sizeof(int));
  }
}
```

Listing 1.4: Interprocedural isolation of the outermost loop of a Finite Impulse Response.

The body of the new `KERNEL` function is given in Figure 1.5. The extra offset parameters are used to perform the translation on the array parameters. The same scheme applies for multidimensional arrays, with one offset per dimension.

```
void KERNEL(int i, int n, int k, int SRC[k], int DST[1],
    int M[k], int SRC_offset, int DST_offset, int M_offset) {
 int v=0;
 for( int j = 0; j < k; ++j )
  v += SRC[i+j-SRC_offset]*M[j-M_offset];
 DST[i-SRC_offset]=v;
}
```

Listing 1.5: Isolated version of the kernel function of a Finite Impulse Response.

4 Redundant Transfer Elimination

The *statement isolation* pass considers a statement independently of its context. However, it is sometimes possible to limit the volume of transferred data when considering the context, either through the elimination of redundant data transfers between isoalted statements, or through overlapping of transfers with computations.

This section informally describes an original contribution to the former using a step-by-step propagation of the memory transfers across the Control Flow Graph (CFG) of the host program. It has been more formally described with proofs in [14]. The main idea is to move *load* operations upward in the Hierarchical Control Flow Graph (HCFG) so that they are executed as soon as possible, while *store* operations are symmetrically moved so that they are executed as late as possible. Redundant load-store elimination is performed in the meantime.

In the following, we only consider optimization of mutliple isolated section during a sequential execution.

4.1 Interprocedural propagation

When a *load* is performed at the entry point of a function, it may be interesting to move it at the call sites. However, this is valid only if the memory state before the call site is the same as the memory state at the function entry point, that is, if there is no write effect during the effective parameter evaluation. In that case, the *load* statement can be moved before the call sites, after backward translation from formal parameters to effective parameters.

Similarly, if the same *store* statement is found at each exit point of a function, it may be possible to move it past its call sites. Validity criteria include that the *store* statement depends only on formal parameters and that these parameters are not written by the function. If this the case, the *store* statement can be removed from the function call and added after each call site after backward translation of the formal parameters.

4.2 Combining load and store elimination

In the meanwhile, the intraprocedural and interprocedural propagation of DMA may trigger other optimization opportunities. *Load*s and *store*s may for instance interact across loop iterations, when the loop body is surrounded by a load and a store; or when a kernel is called in a function to produce data immediately consumed by a kernel hosted in another function, and the DMA have been moved in the calling function.

The optimization then consists in removing *load* and *store* operations when they meet. This relies on the following property: considering that the statement denoted by "`memcpy(a,b,10*sizeof(in))`" is a DMA and its reciprocal is denoted by "`memcpy(b,a,10*sizeof(in))`", then in the sequence `memcpy(a,b,10*sizeof(in ));memcpy(b,a,10*sizeof(in))`, the second call can be removed since it would not change the values already stored in a.

Figure 1.6, illustrates the result of the algorithm. It demonstrates the inter-procedural elimination of data communications represented by the `memload` and `memstore` functions. These function calls are first moved outside of the loop, then outside of the `bar` function; finally, redundant *load*s are eliminated.

```
void bar(int i, int j[2], int k[2]) {
  while (i-->=0) {
    memload(k, j, sizeof(int)*2);
    k[0]++;
    memstore(j, k, sizeof(int)*2);
  }
}
void foo(int j[2], int k[2]) {
  bar(0, j, k);
  bar(1, j, k);
}
```

⇓

```
void bar(int i, int j[2], int k[2]) {
   while (i-->=0) k[0]++;
}
void foo(int j[2], int k[2]) {
  memload(k, j, sizeof(int)*2); // load moved before call
  bar(0, j, k);
  memstore(j, k, sizeof(int)*2);// redundant load eliminated
  bar(1, j, k);
  memstore(j, k, sizeof(int)*2);// store moved after call
}
```

Listing 1.6: Illustration of the redundant load store elimination algorithm.

4.3 Optimizing a Tiled Loop Nest

Alias et al. have published an interesting study about fine grained optimization of communications in the context of Field Programmable Gate Array (FPGA) [1,2,3]. The fact that they target FPGAs changes some considerations on the memory size: FPGAs usually embed a very small memory compared to the many gigabytes available in a GPU board. The proposal from Alias et al. focuses on optimizing loads from Double Data Rate (DDR) in the context of a tiled loop nest, where the tiling is done such that tiles execute sequentially on the accelerator while the computation inside each tile can be parallelized.

While their work is based on the Quasi-Affine Selection Tree (QUAST) abstraction, this section shows how their algorithm can be used with the less expensive convex array region abstraction.

The classical scheme proposed to isolate kernels would exhibit full communications as shown in Figure 1.7. An inter-iteration analysis allows avoiding redundant communications and produces the code shown in Figure 1.8. The

```
for( int i = 0; i < N; ++i ) {
 memcpy(M,m,k*sizeof(int));
 memcpy(&SRC[i],&src[i],k*sizeof(int));
 kernel(i, n, k, SRC, DST, M);
 memcpy(&dst[i],&DST[i],1*sizeof(int));
}
```

Listing 1.7: Code for FIR function from figure 1.1 with naive communication scheme.

```
for( int i = 0; i < N; ++i ) {
 if(i==0) {
   memcpy(SRC,src,k*sizeof(int));
   memcpy(M,m,k*sizeof(int));
 } else {
   memcpy(&SRC[i+k-1],&src[i+k-1],1*sizeof(int));
 }
 kernel(i, n, k, SRC, DST, m);
 memcpy(&dst[i],&DST[i],1*sizeof(int));
}
```

Listing 1.8: Code for FIR function with communication after the inter-iterations redundant elimination.

inter-iteration analysis is performed on a do loop, but with the array regions. The code part to isolate is not bound by static control constraints.

The theorem proposed for exact sets in [1] is the following: [3]

Theorem 1.

$$Load(T) = \mathcal{R}(T) - \left(\mathcal{R}(t < T) \bigcup \mathcal{W}(t < T)\right) \quad (1)$$

$$Store(T) = \mathcal{W}(T) - \mathcal{W}(t > T) \quad (2)$$

where T represents a tile, $t < T$ represents the tiles scheduled for execution before the tile T, and $t > T$ represents the tiles scheduled for execution after T. The denotation $\mathcal{W}(t > T)$ corresponds to $\bigcup_{t>T} \mathcal{W}(t)$.

In Theorem 1, a difference exists for each loop between the first iteration, the last one, and the rest of the iteration set. Indeed, the first iteration cannot benefit from reuse from previously transferred data and has to transfer all needed data, while the last one has to schedule a transfer for all produced data. In other words, $\mathcal{R}(t < T)$ and $\mathcal{W}(t < T)$ are empty for the first iteration while $\mathcal{W}(t > T)$ is empty for the last iteration.

For instance, in the code presented in Figure 1.7, three cases are considered: $i = 0$, $0 < i < N - 1$ and $i = N - 1$.

[3] Regions are supposed exact here; the equation can be adapted to under- and over-approximations.

Using the array region abstraction available in PIPS, a refinement can be carried out to compute each case, starting with the full region, adding the necessary constraints and performing a difference.

For example, the region computed by PIPS to represent the set of elements read for array `src`, is, for each tile (here corresponding to iteration `i`)

$$\mathcal{R}(i) = \{\mathrm{src}[\phi_1] \mid i \leq \phi_1 \leq i + k - 1,\ 0 \leq i < N\}$$

For each iteration `i` of the loop except the first one (here $i > 0$), the region of `src` that is read minus the elements read in all previous iterations $i' < i$ has to be processed; that is, $\bigcup_{i'} \mathcal{R}(i' < i)$.

$\mathcal{R}(i' < i)$ is built from $\mathcal{R}(i)$ by renaming i as i' and adding the constraint $0 \leq i' < i$ to the polyhedron:

$$\mathcal{R}(i' < i) = \{\mathrm{src}[\phi_1] \mid i' \leq \phi_1 \leq i' + k - 1,\ 0 \leq i' < i,\ 1 \leq i < N\}$$

i' is then eliminated to obtain $\bigcup_{i'} \mathcal{R}(i' < i)$:

$$\bigcup_{i'} \mathcal{R}(i' < i) = \{\mathrm{src}[\phi_1] \mid 0 \leq \phi_1 \leq i + k - 2,\ 1 \leq i < N\}$$

The result of the subtraction $\mathcal{R}(i > 0) - \bigcup_{i'} \mathcal{R}(i' < i)$ leads to following region:[4]

$$\mathit{Load}(i > 0) = \{\mathrm{src}[\phi_1] \mid \phi_1 = i + k - 1,\ 1 \leq i < N\}$$

This region is then exploited for generating the *load*s for all iterations but the first one. The resulting code after optimization is presented in Figure 1.8. While the naive version loads $i \times k \times 2$ elements, the optimized version exhibits loads only for $i + 2 \times k$ elements.

5 Applications

The transformations introduced in this article have been used as basic blocks in compilers targeting several different hardware, showing their versatility. They are partially listed here with references to more detailed paper about each work.

- the redundant load store elimination described in Section 4 has been used in [14] for vector instruction sets to optimize loads and stores between vector registers and the main memory. In that case data transfers were not generated by *statement isolation* but through vector instruction packing, leading to the code in Listing 1.9 for a vectorized scalar product. Redundant load store elimination leads to the optimized version in Listing 1.9.
- The communication generation for an image-processing accelerator, TERAPIX [8], described in [14] relies on the *statement isolation* from Section 3.

[4] As the write regions are empty for `src`, this corresponds to the loads.

```
for(i0 = 0; i0 <= 199; i0 += 4) {
  SIMD_LOAD_V4SF(vec20, &c[i0]);
  SIMD_LOAD_V4SF(vec10, &b[i0]);
  SIMD_MULPS(vec00, vec10, vec20);
  SIMD_STORE_V4SF(vec00, &pdata0[0]);
  SIMD_LOAD_V4SF(vec30, &RED0[0]);
  SIMD_ADDPS(vec30, vec30, vec00);
  SIMD_STORE_V4SF(vec30, &RED0[0]);
}
```

```
SIMD_LOAD_V4SF(vec30, &RED0[0]);
for(i0 = 0; i0 <= 199; i0 += 4) {
  SIMD_LOAD_V4SF(vec20, &c[i0]);
  SIMD_LOAD_V4SF(vec10, &b[i0]);
  SIMD_MULPS(vec00, vec10, vec20);
  SIMD_STORE_V4SF(vec00, &pdata0[0]);
  SIMD_ADDPS(vec30, vec30, vec00);
  SIMD_STORE_V4SF(vec30, &RED0[0]);
}
```

Listing 1.9: Body of a vectorized scalar product, before and after redundant load store elimination.

- The SCALOPES project associated an asymmetric MP-SoC with cores dedicated to task scheduling, to a semi-automatic parallelization workflow. *Statement isolation* has been used to generate inter-tasks communications [24].
- SMECY is an innovative compilation tool-chain for embedded multi-core architectures. This on-going project [22] is another use case that exhibits how convex array regions are well suited to communication and mapping problems. In that case, *statement isolation* generates data transfers between different fields of a structure, showcasing that it does not support only arrays, but also imbrication of structure of arrays.
- The code generation for GPUs in Par4All [21] relies on *statement isolation* to efficiently manage communications. It relies on generic data transfers and kernel calls that can use a CUDA or OpenCL backend. A typical output is showcased in Listing 1.10.

```
P4A_copy_to_accel_2d(sizeof pt[0][0], 90, 99, 90, 99 ,0 ,0 ,
    pt ,*p4a_pt0);
P4A_copy_to_accel_1d(sizeof t[0], 20, 20, 0, t, *p4a_t0 );
p4a_launcher_run(*p4a_pt0, range, step, *p4a_t0, xmin, ymin);
P4A_copy_from_accel_2d( sizeof pt[0][0], 90, 99, 90, 99, 0,
    0, pt, *p4a_pt0);
```

Listing 1.10: Typical Par4All-generated DMA.

All these architectures use a load-work-store paradigm, so the code transformations described in this paper can be used to generate or optimized generic data transfers, although they are rather different targets.

6 Related Works

The issue of generating memory transfers between a host processor and an attached accelerator has been studied at multiple occasions in the past.

Convex array regions were already used in the PIPS framework [9] for High Performance Fortran (HPF) code generation. We leverage this approach by decoupling analysis, transfer generation and transfer optimization.

In the same context, the Omega project [20] relied on the manipulation of sets of affine constraints over integer variables. Non-affine conditions and function calls were handled by *uninterpreted function symbols*, a technique described in [28] that does not provide the summarizing capability of interprocedural convex array regions.

Beyond HPF, in the field of embedded computing, other approaches based on memory layout detection and interaction with the memory access patterns have been proposed [16]. The code generation for transfer instructions depending on available communication models has been studied through the polyhedral model [17].

Recently, polyhedral techniques have been applied to generate data communications between a CPU and a GPU, as detailed in [6,18]. The benefit of using convex array regions over these approaches is their ability to retain some important information concerning data accesses even in non-affine situations, by gracefully degrading their accuracy.

An approach that shares some similarities with ours is described in [7]. This paper enhances classical polyhedral techniques to tackle `while` loops and arbitrary conditionnals, relying on over-approximation of the iteration domains through convex hulls. However, it does not propose any solution other than inlining to handle function calls.

7 Conclusion

Automatic code generation currently seems a good lasting option while heterogeneous architectural models are emerging at a sustainable pace, and as a single application may have to be executed on different numerous targets during its life cycle. In this context, efficiently managing data transfers between different memory spaces is a key issue, usually addressed by restricting the control flow of the application kernels.

In this paper, we introduce several techniques relying on the summarizing power of array region analyzes, to lift these barriers and broaden the input class of applications, without sacrificing the efficiency of the generated code.

These techniques have been implemented in the PIPS compiler infrastructure used by the Par4All tool. They have been successfully used to generate code for

GPGPUs, vector processing units, domain-specific architectures, including heterogeneous architectures with task scheduling dedicated cores... Other targets are yet being considered such as multi-GPUs architectures. In addition, our approach could be adapted to directly manage memory hierarchies like software managed cache in GPUs.

Acknowledgments

This work has been supported by French National Research Agency (ANR) through the FREIA Project, the OpenGPU project, and the MediaGPU project. We are grateful to François Irigoin, Ronan Keryell, and Fabien Coelho for their valuable advices.

References

1. Alias, C., Darte, A., Plesco, A.: Program Analysis and Source-Level Communication Optimizations for High-Level Synthesis. Rapport de recherche RR-7648, INRIA (Jun 2011), `http://hal.inria.fr/inria-00601822`
2. Alias, C., Darte, A., Plesco, A.: Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA. In: 2nd International Workshop on Polyhedral Compilation Techniques, Impact (January 2012)
3. Alias, C., Darte, A., Plesco, A.: Optimizing Remote Accesses for Offloaded Kernels: Application to High-level Synthesis for FPGA. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. pp. 1–10. PPoPP, ACM, New York, NY, USA (2012)
4. Amini, M., Coelho, F., Irigoin, F., Keryell, R.: Static compilation analysis for host-accelerator communication optimization. In: International Workshop on Languages and Compilers for Parallel Computing. LCPC (Sep 2011)
5. Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J.O., Pasquier, F.X., Péan, G., Villalon, P.: Par4All: From convex array regions to heterogeneous computing. In: 2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)
6. Baskaran, M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta, R. (ed.) Compiler Construction. Lecture Notes in Computer Science, vol. 6011, pp. 244–263. Springer, Berlin, Heidelberg (Mar 2010)
7. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the International Conference on Compiler Construction. CC, Springer-Verlag, Paphos, Cyprus (Mar 2010)
8. Bonnot, P., Lemonnier, F., Edelin, G., Gaillat, G., Ruch, O., Gauget, P.: Definition and SIMD implementation of a multi-processing architecture approach on FPGA. In: Design Automation and Test in Europe. pp. 610–615. DATE, IEEE Computer Society Press (2008)
9. Coelho, F.: Étude de la Compilation du High Performance Fortran. Ph.D. thesis, Université Paris VI (1993)

10. Creusillet, B., Irigoin, F.: Exact vs. approximate array region analyses. In: Languages and Compilers for Parallel Computing. pp. 86–100. No. 1239 in Lecture Notes in Computer Science, Springer-Verlag (Aug 1996)
11. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. International Journal of Parallel Programming 24(6), 513–546 (1996)
12. Creusillet, B.: Array Region Analyses and Applications. Ph.D. thesis, MINES ParisTech (1996)
13. Entreprise, C.: HMPP workbench. http://www.caps-entreprise.com/hmpp.html
14. Guelton, S.: Building Source-to-Source compilers for Heterogenous targets. Ph.D. thesis, Télécom Bretagne (2011)
15. Guelton, S.: Transformations for memory size and distribution. [14], chap. 6
16. Kandemir, M., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. In: Computer-Aided Design of Integrated Circuits and Systems. vol. 23, pp. 243–260. IEEE (Feb 2004)
17. Meister, B., Leung, A., Vasilache, N., Wohlford, D., Bastoul, C., Lethin, R.: Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In: Workshop on Asynchrony in the PGAS Programming Model. APGAS, Yorktown Heights, New York (Jun 2009)
18. Meister, B., Vasilache, N., Wohlford, D., Baskaran, M.M., Leung, A., Lethin, R.: R-Stream compiler. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1756–1765. Springer (2011)
19. NVIDIA, Cray, PGI, CAPS: The OpenACC Specification, version 1.0 (Nov 2011), http://www.openacc-standard.org/Downloads/OpenACC.1.0.pdf
20. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Conference on Supercomputing. pp. 4–13. Supercomputing, ACM, New York, NY, USA (1991)
21. Silkan: Par4All initiative for automatic parallelization. http://www.par4all.org (2010)
22. Torquati, M., Vanneschi, M., Amini, M., Guelton, S., Keryell, R., Lanore, V., Pasquier, F.X., Barreteau, M., Barrère, R., Petrisor, C.T., Lenormand, É., Cantini, C., De Stefani, F.: An innovative compilation tool-chain for embedded multi-core architectures. In: Embedded World Conference (Feb 2012)
23. Triolet, R., Feautrier, P., Irigoin, F.: Direct parallelization of call statements. In: ACM SIGPLAN Symposium on Compiler Construction. pp. 176–185 (1986)
24. Ventroux, N., Sassolas, T., Guerre, A., Creusillet, B., Keryell, R.: SESAM/ Par4All: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation. In: Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools. pp. 9–16. RAPIDO, ACM, New York, NY, USA (2012)
25. Verdoolaege, S., Grosser, T.: Polyhedral Extraction Tool. In: 2nd International Workshop on Polyhedral Compilation Techniques, Impact (January 2012)
26. Wolfe, M.: Implementing the PGI accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 43–50. GPGPU, ACM, New York, NY, USA (2010)
27. Wolfe, M.: Optimizing Data Movement in the PGI Accelerator Programming Model (Feb 2011), online, available at http://www.pgroup.com/lit/articles/insider/v3n1a1.htm
28. Wonnacott, D., Pugh, W.: Nonlinear array dependence analysis. In: Proceedings of the Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (1995)