Polyhedral Methods for Improving Parallel Update-in-Place

Jing Guo
University of Hertfordshire, Hatfield
United Kindgom
Jing.Guo@herts.ac.uk

Robert Bernecky
Snake Island Research,
Toronto
Canada
bernecky@acm.org

Jeyarajan Thiyagalingam
Oxford e-Research Centre,
University of Oxford, Oxford
United Kingdom
jeyan@bcs.org

Sven-Bodo Scholz
Heriot-Watt University,
Edinburgh
United Kingdom
S.Scholz@hw.ac.uk

ABSTRACT

We demonstrate an optimization, denoted as *polyhedral reuse analysis* (PRA), that uses polyhedral methods to improve the analysis of in-place update for single-assignment arrays. The PRA optimization attempts to determine when parallel array operations that jointly define new arrays from existing ones can reuse the memory of the existing arrays, rather than creating new ones. Polyhedral representations and related dependency inference methods facilitate that analysis.

In the context of SAC, we demonstrate the impact of this optimisation using two non-trivial benchmarks evaluated on conventional shared memory machines and on GPUs, obtaining performance improvements of 2–8 times for LU Decomposition and of 2–10 times for Needleman-Wunsch, over the same computations with PRA disabled.

1. INTRODUCTION

Aggregate updates [12] are a well-known problem in implementing efficient computations on large, aggregate data structures, such as arrays in functional languages. The side-effect-free nature of functional languages demands that all assignments, including those to aggregates, are single assignment. Consequently, successive array modifications, at least on a conceptual level, demand the creation of new arrays for every individual modification operation, even if the original and the modified array differ in one element only.

A significant amount of research has gone into attempts to eliminate or reduce this potential source of excessive copying and memory demand. The most effective techniques found thus far are based on advanced forms of reference counting [5, 8, 10]. These techniques combine static analysis and code transformation with dynamic counters that maintain the number of active references to each individual use of every array. This knowledge enables the executing code to avoid memory allocations and copying, in almost all cases. In particular, successive modifications of one individual array can be identified, and then compiled into code of identical efficiency to its imperative counterpart.

While such reference-counting-based techniques suffice for update operations of individual array elements, the situation becomes more challenging when dealing with update operations that permit *parallel updates* of several elements of arrays. By parallel update, we mean update operations in which the semantics of the language does not impose any particular compute order for individual elements of the result array. For such parallel-update operations, memory reuse is a more complex problem. In many cases, increased potential for memory reuse can be gained by imposing a *schedule* – restrictions on the compute order of the individual elements. In our environment, we are not willing to sacrifice any potential parallelism, as we want to target architectures, such as GPUs, that offer high-levels of parallelism with minimal overhead. Consequently, we are exclusively interested in analyses that identify memory-reuse potential, **without** restricting the compute order at all. *I.e.*, the compiler has to ensure that sharing the memory between unmodified and modified arrays does not introduce undesired dependencies.

The Polyhedral Model is known for its ability to transform complex loop nests, as a way to improve the generated code in various ways, such as improving memory usage and locality of reference. Key to this ability is the systematic capture of data dependencies within loop nests, including alternative control paths governed by predicates. This paper shows how polyhedral methods can identify whether parallel updates of array elements can be done in place, without restricting potential execution schedules. We present this work in the context of the functional array programming language SAC (Single Assignment C), and we evaluate its impact on sequential execution for conventional shared-memory systems and for parallel execution on GPUs. We feel that this work applies equally well in an imperative setting, particularly when polyhedral techniques for code generation are used.

The paper is structured as follows: Section 2 expands on the problem of data-parallel update-in-place. It provides the necessary background in SAC, its data-parallel update operations and the basic reference counting technique that is used. Section 3 explains how we use a polyhedral representation for update-in-place analysis optimization. It sketches

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://impact.gforge.inria.fr/impact2014

the analysis algorithm and its implementation using existing polyhedral tools. Section 4 discusses the performance impact of this analysis on two benchmarks on conventional CPU-based systems and on highly parallel GPU systems. Section 5 relates our work to existing work; Section 6 summarizes the salient features and impact of our approach, and its potential relevance to domain-specific languages (DSL).

2. UPDATE-IN-PLACE CHALLENGE

The programming language SaC combines a Matlab-like array programming style with support for high-performance execution on parallel platforms, such as shared-memory multi-core systems or GPU accelerated systems (see *e.g.* [18,19]). The transformation of high level, generic code in SaC into efficient, multi-threaded executables for various parallel platforms hinges, to a large degree, on the functional foundations of SaC. The side-effect-free setting of SaC guarantees that all data dependencies are explicit. Moreover, it lays the foundation for all concurrency being directly derivable.

However, a side-effect-free setting comes at a price: a fundamental challenge of that setting is that modification operations are forbidden. Instead, they are interpreted as the creation of new objects from existing ones, even though those objects may differ in only a few elements. Apparently imperative programs, such as this SaC code:

```
1   a = [1,1,1,1,1];
    a[3] = 42;
3   return(a);
```

must, in fact, be considered as two variable definitions:

```
1   a = [1,1,1,1,1];
    a' = modarray( a, [3], 42);
3   return(a');
```

where the conceptually different arrays, `a` and `a'`, are explicitly defined. When compiling such code for high-performance execution, it is paramount to find ways to project the allocation of `a` and `a'` into the same memory space and, thereby, to end up with a runtime update-in-place operation, in much the same way as a naive imperative interpretation of the above code would suggest. Key to being able to do this kind of optimisation is a sharing analysis of variables, to keep track of the number of references to a given data object. Advanced reference-counting techniques, such as those pioneered in the context of SISAL [5,9], and later refined in the context of SaC [10], deliver this information, in most cases, at compile time and in the few remaining cases, at runtime.

While these techniques suffice in the example given above, the situation becomes more involved when looking at data-parallel update operations that concurrently modify more than one element of an array. Consider a straightforward extension of the previous example to a two-dimensional case. In SaC , we can modify entire rows of a matrix by means of a single assignment:

```
1   a = [  [1,2,3]
          ,[4,5,6]  ];
3   a[0] = [42,42,42];
    return(a);
```

This replaces the entire first row of the initial 2×3 matrix `a` by a 3-element vector of values 42. As in the scalar case above, our compiler will identify that the second version of `a`, defined through the modarray operation, can, in fact, reuse the original matrix `a`, thereby saving a second allocation as well as the need to copy the array row values `[4,5,6]`.

Things are more complicated when the values to be modified depend on existing entries in the same array. An example of this is an element-wise increment, such as:

```
    a = [  [1,2,3]
2         ,[4,5,6]  ];
    a[0] = [a[0,0]+1,a[0,1]+1,a[0,2]+1];
4   return(a);
```

This can reuse the memory of `a`, as all reads are *local*, *i.e.*, it reads only from the same position that is to be modified. Now consider a slight variant of the previous example:

```
    a = [  [1,2,3]
2         ,[4,5,6]  ];
    a[0] = [a[0,2],a[0,1],a[0,0]];
4   return(a);
```

In this example, we reverse the order of the elements of the first row. Here, the non-local read accesses in line 3 into the matrix `a` defined in line 1 inhibit a direct reuse of the memory of `a`. The only way to reuse the memory of `a` in this example would be to store the read values in an intermediate location before performing the in-place updates.

Interestingly, there are situations where we can do without intermediate storage, despite having non-local read accesses. Consider yet another slight modification of our example:

```
    a = [  [1,2,3]
2         ,[4,5,6]  ];
    a[0] = [a[1,2],a[1,1],a[1,0]];
4   return(a);
```

The only difference from the previous example is that elements from the second row are placed in reversed order into the first row. The values for updates are read from an area of the argument arrays that is identical to the corresponding area of the result array, so no intermediate structure is needed to hold those elements.

From these few examples, we learn that, for memory reuse of parallel update operations: *The ability to map the argument array `a` and the result array `a` into a single memory location depends on two things: As in the non-parallel, single-element-update case, it depends on the number and location of references to `a`.* **In addition**, *it depends on the index range which the parallel update reads from. If all those reads are local or fall into a region that remains unmodified, sharing the memory does not impose new dependencies.*

Parallel array update functions in SaC are defined via its data-parallel construct, the WITH-loop, including the `modarray` operation seen in the above examples. WITH-loops define new arrays from existing ones, in a map-like, parallel fashion.

2.1 Running Example

Figure 1 gives a SaC running example WITH-loop in which a 60-element vector `d` is defined from vectors `a`, `b`, and `c`; Figure 2 shows an equivalent C version. The WITH-loop contains three different index ranges, with separate definitions. For the first 20 elements (indices `0` to `19`), the values are copied from vector `a`; for the last 20 elements the values are copied from vector `c`. The 20 middle elements are alternately copied from `b`, or computed as a sum from one element of `a`, `b`, and `c`, as depicted at the bottom of Figure 1.

All WITH-loops are inherently data-parallel: their semantics guarantees that all elements of the result can be computed in parallel. Furthermore, single assignment semantics requires that the array `d`, defined here, *cannot* be referenced on the right-hand side of the assignment. The individual

```
d = with {
      ( [ 0] <= [i] < [20]) : a[i];
      ( [20] <= [i] < [40]) :
          (( i % 2 ) == 0) ?
            b[i] :
            a[i-20] + b[i-1] + c[i+10];
      ( [40] <= [i] < [60]) : c[i];
    } : genarray( [60], 0);
```

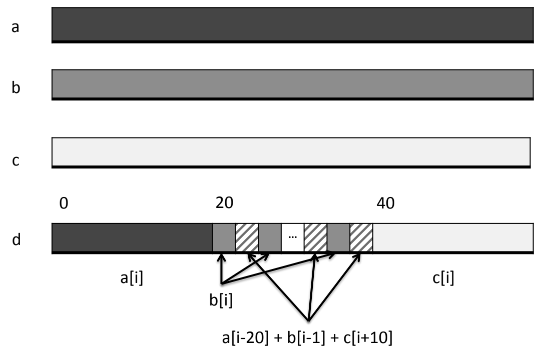


Figure 1: Running example: vector `d = f(a, b, c)`

index ranges are currently restricted to rectangular grids of potentially statically unknown sizes and stepping, but could, in principle, be extended to allow for arbitrary polyhedra. Missing indices within the result array are filled with the default element, `0` in this example.

We allow programmers to specify overlapping index ranges, but internally split those WITH-loops into non-overlapping ones, then use an optimisation to fold, whenever possible, those WITH-loops back into WITH-loops with multiple, non-overlapping index ranges. This guarantees that the individual index ranges of all WITH-loops are non-overlapping, before doing any analysis of memory layout. More details on the semantics of WITH-loops in SAC , and on how non-overlapping ranges are enforced can be found in [16].

As in the modarray example above, WITH-loops define new arrays. For this paper, we furthermore assume that the layout of the result in memory is given and cannot be changed. We are interested in finding out whether any one of the arrays from which some values are being copied into `d` can be updated in place. This eliminates the need to allocate and deallocate memory; more importantly, it eliminates the need to copy those elements from one array into another.

Looking more closely at our running example, we observe that more than one array could serve as what we call a *reuse candidate*: Besides the obvious candidate `a`, the vector `b` could be reused. For both arrays, all read accesses are to elements that are copied in an unmodified fashion.

In contrast, the vector `c` cannot serve as a reuse candidate: some read accesses into `c` overlap with some of the non-copying write accesses into `d`, *e.g.*, the computation of `d[21]`, which refers to `c[31]`.

The central contribution of this paper is to define an analysis based on polyhedral methods that identifies memory-reuse candidates for arrays that hold the results of WITH-loops. This analysis, combined with pre-existing reference-counting techniques, enables extended reuse of memory; it also reduces the need to copy unmodified parts of arrays.

3. POLYHEDRAL REUSE ANALYSIS

In the previous section, we identified that it is key to this optimisation to be able to analyse index spaces, their mappings through access functions and their intersections. The polyhedral model offers a concise and suitable representation for these aspects; it also comes with readily available tools for computing such mappings and intersections. We now show how we map our WITH-loops into the polyhedral setting, then we formulate our analysis in terms of such a polyhedral representation.

3.1 A Polyhedral Formulation

When switching to a polyhedral formulation, we map WITH-loops into an equivalent representation, with loops using explicit read and write accesses. Every WITH-loop represents a sequence of loop nestings of identical nesting depth, given by the length of the index vector (`[i]` in Figure 1). The number of subsequent nestings is given by the number of definitions within the WITH-loop. In our example we have three definitions ranging over 20 elements each. The innermost bodies of the nestings contain assignments to the result array at the corresponding position. The resulting representation of our example WITH-loop as standard FOR-loops is shown in Figure 2.

```
d = (int *)malloc( 60 * sizeof( int ));

for( i=0; i<20; i++)
  d[i] = a[i];

for( i=20; i<40; i++)
  if( ( i % 2 ) == 0) {
    d[i] = b[i];
  } else {
    d[i] = a[i-20] + b[i-1] + c[i+10];
  }

for( i=40; i<60; i++)
  d[i] = c[i];
```

Figure 2: Running example in C.

In general, the loop nests generated from WITH-loops have several invariants that stem from their semantics and that cannot be compromised even though WITH-loops can contain arbitrary SAC expressions, including conditionals, loops, or WITH-loops: The data-parallel nature of WITH-loops guarantees that all write accesses into a result array appear as the last operation within all loop nests. It is also guaranteed that the iteration space and the result array data space have a one-to-one correspondence, ensuring that every element is written to exactly once.

We first formalise our analysis requirements in terms of conventional loop representations. Since WITH-loops can be defined in a nested fashion, we distinguish between the *local* and *global* write accesses of each WITH-loop $\mathcal{W}l$ represented by conventional loops: local write accesses refer to arrays defined inside the body of a $\mathcal{W}l$, whereas global ones refer to the result of the $\mathcal{W}l$ itself. For our purposes, we are only interested in global write accesses, since they determine whether $\mathcal{W}l$ can be performed in-place or not.

An *in-place read* of a WITH-loop is an array access $\mathcal{A}[\mathcal{I}v]$ which resides at the same loop level as a global write access $\mathcal{A}'[\mathcal{I}v]$. If array $\mathcal{A}$ is reused to be destructively updated by the WITH-loop, *i.e.*, $\mathcal{A} = \mathcal{A}'$, both accesses will reference the same memory location during each iteration of the enclosing

WITH-loop nest, causing a *loop-independent* anti-dependence from $\mathcal{A}[\mathcal{I}v]$ to $\mathcal{A}'[\mathcal{I}v]$. However, since the write is always performed after the read, that dependence is guaranteed to be preserved, irrespective of the relative execution order of different iterations. Therefore, in-place reads can never prevent the reuse of their accessed arrays for in-place update. In Figure 2, the accesses in lines 4, 8, and 14 are all in-place reads since they are at the same loop levels and have the same index vectors as the respective global write accesses.

A *copy assignment* in a WITH-loop is of the form $\mathcal{A}[\mathcal{I}v] = \mathcal{A}'[\mathcal{I}v]$ where $\mathcal{A}[\mathcal{I}v]$ is a global write access and $\mathcal{A}'[\mathcal{I}v]$ is an in-place read. Essentially, it performs data copying from $\mathcal{A}'$ to $\mathcal{A}$ in an element-wise manner. All three assignments with in-place reads in our example are of this nature.

With this nomenclature at hand, we can formulate our problem as follows:

An array $\mathcal{A}$ is a reuse candidate iff the iteration spaces associated to all non-in-place reads from $\mathcal{A}$ map into iteration spaces of copy assignments with in-place reads from $\mathcal{A}$.

3.2 Identifying Reuse Candidates

A formalisation of the in-place update analysis is shown in Algorithm 1. Given a WITH-loop $\mathcal{W}l$, the algorithm infers the set of reuse candidates whose memory can be reused for destructive update by $\mathcal{W}l$. Each array access $\mathcal{A}[\mathcal{I}v]$ in $\mathcal{W}l$ is associated with an iteration vector $\mathcal{I}$ and a control path set **CP**. $\mathcal{I}$ describes the polyhedron that $\mathcal{I}v$ ranges over and **CP** is a normalised form of all predicates that dominate that particular array access. **CP** contains a set of lists of simple predicates, representing a disjunctive normal form of these predicates. **CP** may contain only one empty path if $\mathcal{A}[\mathcal{I}v]$ is not in any conditionals. Depending on the type of $\mathcal{A}[\mathcal{I}v]$, one of two different accessed data spaces is computed:

- If $\mathcal{A}[\mathcal{I}v]$ is a non-in-place read, the data space $\mathcal{DS}$ it reads under the enclosing WITH-loops and conditionals is computed. The total read data space of $\mathcal{A}$ (stored in table **RDS**) is then updated by taking its union with $\mathcal{DS}$. In-place reads are ignored, as they can never be reuse-preventing, as discussed previously.
- If $\mathcal{A}[\mathcal{I}v]$ is the global write access of a copy assignment $\mathcal{A}[\mathcal{I}v] = \mathcal{A}'[\mathcal{I}v]$, the data space $\mathcal{DS}$ of $\mathcal{A}'$ copied by this assignment under the enclosing WITH-loops and conditionals is computed. The total copy data space of $\mathcal{A}'$ (stored in table **CDS**) is then updated by taking its union with $\mathcal{DS}$.

To compute $\mathcal{DS}$ for $\mathcal{A}[\mathcal{I}v]$, the index vector $\mathcal{I}v$, iteration vector $\mathcal{I}$ and control path set **CP** of $\mathcal{A}[\mathcal{I}v]$ are passed to Algorithm 2, which analyses whether $\mathcal{A}[\mathcal{I}v]$ constitutes an affine access within the given iteration space and symbolic constants. That algorithm calls procedure `Access_Analysis`, which returns an array access function $\mathcal{F}$, represented by a suitable matrix, or *NULL* if an affine access cannot be determined.

If such a matrix $\mathcal{F}$ is found, we compute the relevant portion of the iteration domain by extending the affine description of the iteration domain with the constraints of the individual control paths. The impact of a particular control path in **CP** on the iteration vector is computed by a call to the `Domain_Analysis` procedure, resulting in a potentially restricted sub-domain $\mathcal{ID}$. Each of these sub-domains are then mapped into the array access domains by computing

Algorithm 1: In-place Update Analysis

```
Input: A WITH-loop $\mathcal{W}l$ which the in-place update
       analysis is performed upon;
1  Let RDS be a (initially empty) table with pairs
   ($\mathcal{A} \mapsto \mathcal{RDS}$) where $\mathcal{A}$ is an array and $\mathcal{RDS}$ is its data
   space read by $\mathcal{W}l$;
2  Let CDS be a (initially empty) table with pairs
   ($\mathcal{A} \mapsto \mathcal{CDS}$) where $\mathcal{A}$ is an array and $\mathcal{CDS}$ is its data
   space copied by $\mathcal{W}l$;
3  Let the vector of symbolic constants referred to in $\mathcal{W}l$
   be $\mathcal{SC}iv = [sc_1 \dots sc_m]$;
4  foreach Array access $\mathcal{A}[\mathcal{I}v]$ in $\mathcal{W}l$ do
5      Let $\mathcal{I}$ be the iteration vector associated with this
       access;
6      Let CP be the set of control paths associated with
       this access;
7      if $\mathcal{A}[\mathcal{I}v]$ is a non-in-place read access then
8          $Affine, \mathcal{DS} \leftarrow$ Get_Data_Space($\mathcal{I}, \mathcal{I}v$, CP,
           $\mathcal{SC}iv$);
9          if $Affine = True$ then
10             RDS[$\mathcal{A}$] $\leftarrow$ RDS[$\mathcal{A}$] $\cup \mathcal{DS}$;
11         else
12             Terminate;
13     else if $\mathcal{A}[\mathcal{I}v]$ is the global write access in a copy
       assignment then
14         Let $\mathcal{A}'[\mathcal{I}v]$ be the in-place read of this copy
           assignment;
15         $Affine, \mathcal{DS} \leftarrow$ Get_Data_Space($\mathcal{I}, \mathcal{I}v$, CP,
           $\mathcal{SC}iv$);
16         if $Affine = True$ then
17             CDS[$\mathcal{A}'$] $\leftarrow$ CDS[$\mathcal{A}'$] $\cup \mathcal{DS}$;
18         else
19             Terminate;
20     else
21         Continue;
22 RC $\leftarrow \emptyset$;
23 foreach $\mathcal{A}$ with ($\mathcal{A} \mapsto \mathcal{CDS}$) $\in$ CDS do
24     if RDS[$\mathcal{A}$] $\subseteq$ CDS[$\mathcal{A}$] then
25         RC $\leftarrow$ RC $\cup \{\mathcal{A}\}$;
Output: A set of arrays RC that can be reused for
        in-place update by $\mathcal{W}l$.
```

$\mathcal{F} \bullet \mathcal{ID}$. The result is the union of all such array access spaces under different control paths. After all array accesses in $\mathcal{W}l$ have been examined, each entry ($\mathcal{A} \mapsto \mathcal{RDS}$) in table **RDS** represents the total data space of $\mathcal{A}$ that is read by $\mathcal{W}l$, not including those accessed by in-place reads. Similarly, each entry ($\mathcal{A} \mapsto \mathcal{CDS}$) in table **RDS** represents the total data space of $\mathcal{A}$ that is copied by $\mathcal{W}l$.

Algorithm 2: GetDataSpace ($\mathcal{I}$, $\mathcal{I}v$, **CP**, $\mathcal{SC}iv$)

```
Input: A loop iteration vector − $\mathcal{I}$;
       An access index vector − $\mathcal{I}v$;
       A control path set − CP;
       Symbolic constant vector − $\mathcal{SC}iv$;
1  $Affine \leftarrow False$;
2  $\mathcal{DS} \leftarrow NULL$;
3  $\mathcal{F} \leftarrow$ Access_Analysis($\mathcal{I}v$, $\mathcal{I}$, $\mathcal{SC}iv$);
4  if $\mathcal{F} \neq NULL$ then
5      $Affine \leftarrow True$;
6      foreach $\mathcal{CP} \in$ CP do
7          $\mathcal{ID} \leftarrow$ Domain_Analysis($\mathcal{I}$, $\mathcal{CP}$, $\mathcal{SC}iv$);
8          if $\mathcal{ID} \neq NULL$ then
9              $\mathcal{DS} \leftarrow \mathcal{DS} \cup \mathcal{F} \bullet \mathcal{ID}$;
10         else
11             $Affine \leftarrow False$;
12             $\mathcal{DS} \leftarrow NULL$;
13             Break;
Output: $Affine$ − $True$ if the iteration domain and
        access are affine, $False$ otherwise;
        $\mathcal{DS}$ − Data space accessed, $NULL$ if $Affine$
        is $False$;
```

After computing all relevant data spaces, the algorithm infers the set of valid reuse candidates. An array $\mathcal{A}$ is considered only if part of it is copied by $\mathcal{W}l$ (*i.e.*, there is an entry ($\mathcal{A} \mapsto \mathcal{CDS}$) in table **CDS**). This ensures the effectiveness of copy elimination if $\mathcal{A}$ is selected to be reused. Array $\mathcal{A}$ may potentially be a reuse candidate if its read data space is a subset of its copy data space. In other words, if $\mathcal{A}$ is reused for in-place update, every read from $\mathcal{A}$ will return the same result, irrespective of its relative execution order with a write to the same array location. Therefore, iterations can be executed in arbitrary order while preserving determinism.

In our implementation, we build on functions from the POLYLIB library to perform the various operations over polyhedra, such as computing the union of data spaces (*e.g.*, $\cup$), determining the inclusion relationship (*e.g,*, $\subseteq$) and finding the image of a polytope under affine transformation (*i.e.*, $\bullet$).

4. PERFORMANCE IMPACT

We evaluated the effectiveness of our polyhedral-model-based, update-in-place optimization using two benchmarks: *LU Decomposition* and *Needleman-Wunsch* [14], generating sequential and CUDA codes, referred to here as SAC-SEQ and SAC-CUDA . Our experiments were conducted on two Linux64 2.6.35 platforms: The first system, dubbed C1060, comprises an earlier generation of GPU on a 2-core, 1.6G Hz Aeon 5110, L1=32B, L2=4MB; the second system, dubbed GTE480, comprises a GTE480 GPU on a 4-core 2.8G Hz Intel i7, L1=64B, L2=256B, L3=8MB.

4.1 LU Decomposition

The kernel of the SAC implementation of *LU Decomposition* is shown in Figure 3. We observe that both WITH-loops

```
   for( k = 0; k < N-1; k++) {
 2   A = with {
            ( [k+1,k] <= [i,j] < [N,k+1])
 4                                     : A[i,j]/A[k,k];
          } : modarray( A);
 6   A = with {
            ( [k+1,k+1] <= [i,j] < [N,N])
 8                              : A[i,j]-A[i,k]*A[k,j];
          } : modarray( A);
10 }
```

Figure 3: SaC implementation of ***LU Decomposition*****.**

in this algorithm can be performed in place. The only non-

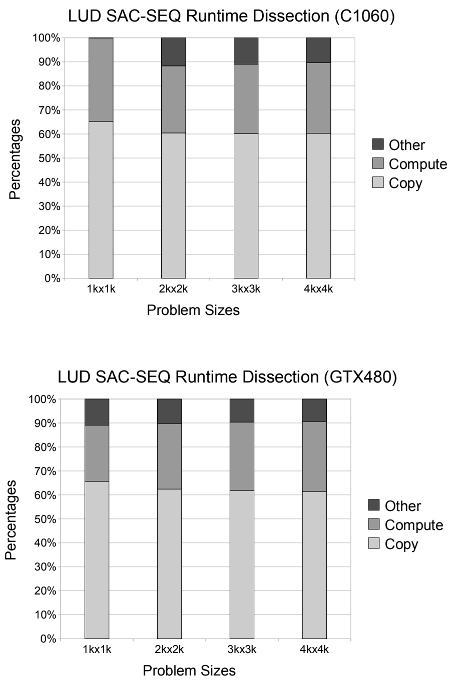


Figure 4: ***LU Decomposition*** **SaC-seq dissections**

in-place read in the first WITH-loop is the selection `A[k,k]` in line 4, which refers to an index outside the specified range. Since we are dealing with a MODARRAY-WITH-loop, all missing elements are inserted as index ranges over copy assignments from `A`. Consequently, our analysis identifies `A` as a reuse candidate for the WITH-loop in lines 2-5. Similarly, the two non-in-place reads in line 8 are identified as reads into copy assignment ranges leading to a further reuse of the memory of `a` for the WITH-loop in lines 6-9. Note here, that our previous technique described in [10] would not consider any reuse possible here due to the occurance of non-in-place reads in both cases.

Figures 4 and 5 show the runtime dissections of SAC-SEQ and SAC-CUDA, without in-place update, on several different problem sizes. The total execution time is divided into three components: compute time of the actual decomposition (i.e. column and sub-matrix computations), data copy-

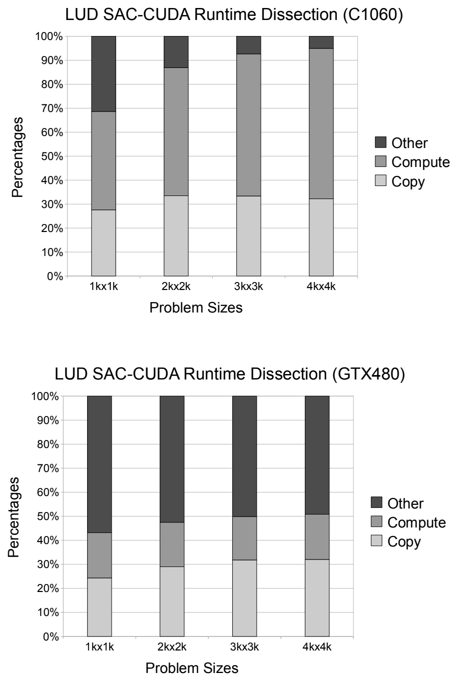


Figure 5: ***LU Decomposition*** **SaC-cuda dissections**

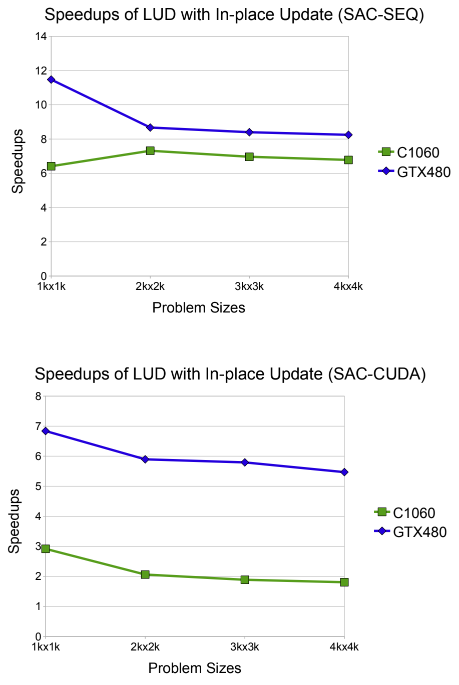


Figure 6: LU speedups using update-in-place

ing time, and other overheads (mainly as a result of the memory management overhead). As can be observed from the dissections of SAC-SEQ runtime, the average percentages of the three components are very consistent across all problem sizes on both platforms at 10% for overheads, 29% for computation and 61% for data copying. The only exception is for 1024×1024 matrix on C1060 where the overheads are almost negligible. This is because allocating or freeing memory of size $1024 \times 1024 \times 8 = 8$MB (assuming `double` data type) is considerably faster than the other sizes on C1060. By contrast, the SAC-CUDA runtime dissections show that the execution components with the highest percentages are computation (54% on average) and overheads (52% on average) on C1060 and GTX480 respectively. The two main factors that cause the reduced significance of data copying are: **(i)** Data copying between arrays in the GPU memory is significantly faster than it is in the CPU memory due to the much higher GPU memory bandwidth. This substantially reduces the absolute data copying time and **(ii)** GPU memory deallocation operation on GTX480 is considerably more expensive than its counterpart on either the host or C1060.

Figure 6 shows the effect on wall-clock runtime when using our polyhedral update-in-place code. On average, the performance of SAC-SEQ is improved by 6.9× and 9.2× on C1060 and GTX480 respectively. These speedups, at first glance, appear to be in contradiction to the dissection graphs shown in Figure 4. Those graphs show that computation takes approximately 29% of total execution time on both platforms. Therefore, an average improvement of only 3.3× is expected if both the overheads and data copying are eliminated by the optimization. Analysis revealed that polyhedral in-place computations improved cache behavior, leading to unexpected performance gains.

By contrast, the performance improvements of SAC-CUDA are more consistent with the corresponding runtime dissection graphs. The average speedups are 2.2× and 6× on C1060 and GTX480 respectively. The absence of cache hierarchy in C1060 means a write will always access the global memory directly regardless of whether the same element has been read before or not. Therefore, in-place update does provide the benefits as those described in the sequential case. In the GTX480, a 768B L2 cache was introduced to provide fast data access for all processing cores (there is also an L1 cache but it is read-only). However, since the cache is often shared among tens of thousands of concurrently executing threads, the cache line loaded due to the access of an element is less likely to be present when the corresponding write is issued. Therefore, write misses may still occur with high frequency, a situation similar to writing to a different array.

Size	SEQ	SEQ +PRA	CUDA	CUDA +PRA	MEM	MEM +PRA
1024	0.28	1.49	0.56	10.21	16	9
2048	0.19	1.33	0.97	11.22	66	34
4096	0.19	1.29	1.15	11.39	258	130

Table 1: Performance of the LUD Benchmark on the Zen platform in GFLOPS and memory use in MB

We show the absolute performance and memory requirements of the LUD benchmark in Table 1 as obtained from the Zen platform (OS: Linux 2.6.35, CPU: Intel X5650i at 2.67GHz, GPU: nVidia C2070, Memory: 24GB, L1:32KB, L2: 256KB and L3:12MB). The +`PRA` columns indicate the performance when the re-use optimisation is enabled.

We can see an almost 7-fold increase in sequential performance reflected in an increase from 190 MFLOPS to 1.3 GFLOPS. On the GPU, we see an even higher improvement from roughly 1 GFLOP to 11 GFLOPS. The memory use decreases by the amount needed for exactly one array of the problem size. Sinze the entire application mainly requires one array of that size only, we see an almost 50% reduction in space demand. However, for all problem sizes the entire memory demand fits into the L2 cache.

4.2 Needleman-Wunsch

The *Needleman-Wunsch* algorithm is a non-linear, global optimisation method for DNA sequence alignments. The wavefront nature of the computation is such that the value of each data element depends on the values of its north-, west- and north-west neighbouring elements.

```
/* Compute upper left triangular matrix */
for( i = 1; i < N; i++) {
  A = with {
          ( [1,1] <= [r,c] < [i+1,i+1])
             : ( r == (i - c + 1) ?
               maximum( A[r-1,c-1] + ref[r,c],
                        A[r,c-1] - penalty,
                        A[r-1,c] - penalty)
               : A[r,c] );
       } : modarray( A);
}

/* Compute lower right triangular matrix */
for( i = 1; i < N; i++) {
  A = with {
          ( [i+1,i+1] <= [r,c] < [N,N])
             : ( r == (N - c + i) ?
               maximum( A[r-1,c-1] + ref[r,c],
                        A[r,c-1] - penalty,
                        A[r-1,c] - penalty)
               : A[r,c]);
       } : modarray( A);
}
```

Figure 7: SaC implementation of *Needleman-Wunsch*.

The performance-relevant kernel of the SaC implementation of the algorithm is shown in Figure 7. The general structure is that we twice have a FOR-loop, within which a recomputation of a diagonal is specified by means of a `modarray`-WITH-loop. However, since WITH-loops can only express rectangular data/iteration space, a conditional statement is required to ensure that only elements on the diagonal are computed. The predicates in both WITH-loops are affine, as they depend only on WITH-loop indices and symbolic constants. Consequently, during each FOR-loop iteration, only the diagonal elements are updated with new values, while the rest of the matrix is copied over to the output array.

A closer examination of the accesses reveals that the computation of each diagonal only depends on data in two other diagonals which are copied from the previous version to the new output matrix. As in the LU example, our in-place analysis successfully identifies `A` as being a reuse candidate and eliminates all overhead that would result otherwise.

Figures 8 and 9 show the runtime dissections of SaC-SEQ and SaC-CUDA in the absence of in-place update. Similar to the LU example, three different components have been measured: compute time, data copying time, and other overheads. The dissection graphs show that the application devotes most of its execution time to data copying (between 46% and 66%) for sequential and parallel execution.

Figure 10 shows the effect of doing in-place updates. The performance of SaC-SEQ is improved by about 3× and 2.6× on the C1060 and GTX480, respectively. The SaC-CUDA benefits are greater, with average speedups of 9.7× and

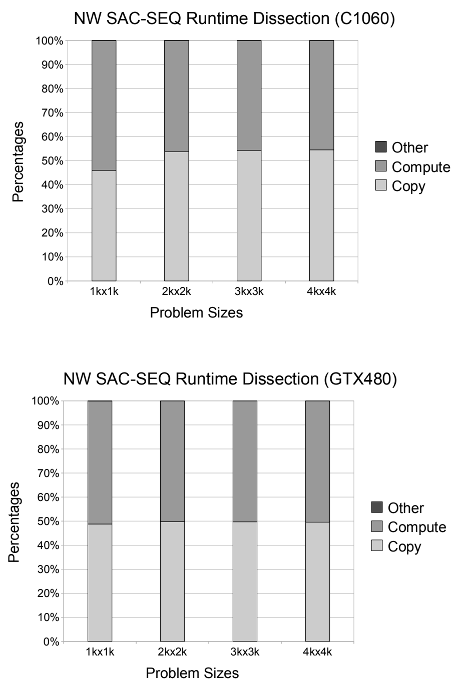


Figure 8: *Needleman-Wunsch* SaC-seq dissections

Size	SEQ	SEQ +pra	CUDA	CUDA +pra	MEM	MEM +pra
1024	1.58	5.95	4.29	47.57	26	18
2048	0.69	2.60	3.72	24.65	98	66
4096	0.35	1.30	2.13	12.59	386	258

Table 2: Performance of the NW Benchmark on the Zen platform in MFLOPS and memory use in MB

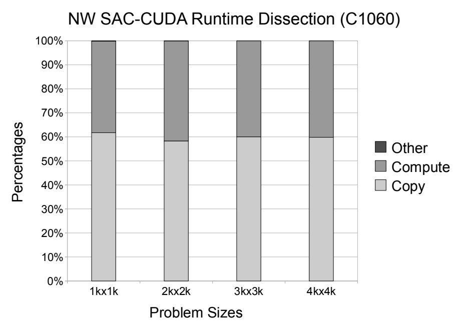
10.3× observed on the two platforms. The disproportionate gains in the GPU setting here are a consequence of conditionals within the parallel code. By identifying the reuse, one of the branches of the conditionals becomes empty, which triggers different, more efficient GPU code generation.

Table 2 shows the absolute performance of the NW benchmark with and without the PRA-optimisation on the Zen platform as well as the application's memory demand.

While we again see the relative improvements reflected in the performance, the overall performance for the benchmark lies in the MFLOPS range rather than reaching into the GFLOPS. This is a consequence of the limitation of the index ranges in WITH-loops to rectangular spaces. If we would support arbitrary polyhedra further improvements would be achieved here. Nevertheless, the improvements due to PRA are clearly reflected in the 3-4 fold improvements on the sequential case and in the 6-11 fold improvements on the GPU. On the memory side we can again observe that we improve by exactly one array of the full size. However, since 2 big arrays are inevitable (`a` and `ref`) we only see a 30% improvement for all sizes.

5. RELATED WORK

Our work directly extends prior work on reference counting for solving the update-in-place problem. Work in SISAL [5, 8, 9] focuses mainly on aliasing analysis and techniques,



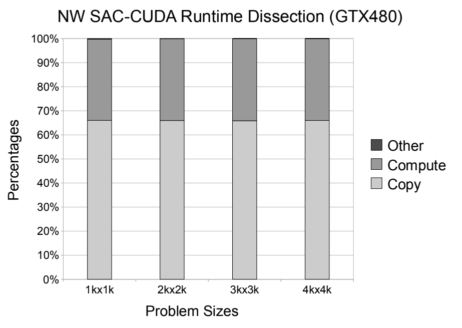
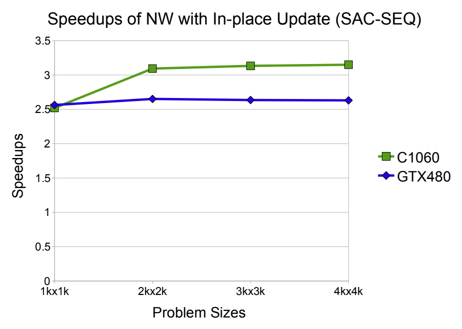


Figure 9: *Needleman-Wunsch* **SaC-cuda dissections**



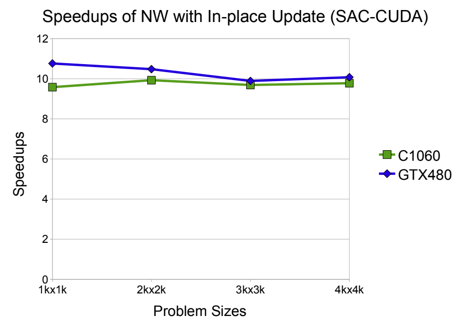


Figure 10: NW speedups with update-in-place

to linearise data-flow graphs so as to maximise reuse potential. Later work, in SAC [10], incorporates parallel update operations that support local read operations. The work presented in this paper extends update-in-place further, by allowing non-local read operations whenever we can prove that those refer to non-modified regions.

Most of the work in the context of polyhedral techniques assumes a far more general setting than the restricted update-in-place problem considered in this paper. Those works generally start by assuming that loop nests may contain arbitrary read and write operations, and that they may have memory-introduced, as well as non-memory-introduced dependencies. In contrast, we start from a single-assignment setting that is guaranteed to be completely dependence-free. This simplifies our analysis considerably, as is often the case when using single-assignment methods.

Much of the polyhedral work is concerned with specific aspects, such as tilings to improve cache locality [4], restructuring data to enable vectorisation of stencil codes [11], or data organisation for creating efficient GPU codes [1–3]. In that context, update-in-place plays little or no role.

Other polyhedral work focuses on the reduction of the memory used, for instance, due to limited memory resources on a given architecture or even to reduce required memory bandwidth. Rather early work, in the context of the programming language ALPHA by Wilde *et al.* [20] and later by Quillere *et al.* [15], applies the polyhedral model to minimise the memory use given a fixed schedule. Starting from a fixed set of variables, they minimise the index spaces used by mapping several subsequent assignments to different locations into multiple assignments of fewer locations, effectively reducing the index spaces required for the variables involved. Similarly, the work by Lefebvre *et al.* [13] and that of de Greef *et al.* [7] aims at reducing memory requirements within a specified schedule.

Later work on reducing memory-introduced dependencies, by Strout *et al.* [17], as well as by Cohen [6], attempts to expand loop nests, then to compress memory demand again, without reintroducing dependencies. While this work is very close to ours in spirit, the setting is vastly different. In our setting, full expansion is a given, as we are always dealing with data-parallel operations. Our main goal is **not** an attempt to reduce storage requirements, but to avoid superfluous copying which has been introduced by the inherently data-parallel construction of the program.

6. CONCLUSIONS

In this paper, we demonstrate how polyhedral methods can be used to improve in-place updates for data-parallel array updates in functional languages. In doing so we vastly deviate from the classical use of the polyhedral approach. Firstly, our starting point is very different. We do not assume arbitrary nestings of loops with potential dependencies. Instead, we have dependency-free single assignment loops as a starting point. Secondly, we do not intend to use the polyhedral framework to transform loop nestings for improved parallelism, locality or minimised representations of intermediate data structures. Instead, we limit the use of the polyhedral approach to representing the access pattern within our code and to conveniently implementing our update-in-place analysis by means of readily available libraries on top of this representation.

For two example codes, we demonstrate the potential benefits of this technique. We look at two different target platforms: sequential shared-memory execution and execution on GPUs. In both cases we see speedups of a factor between 2 and 12. While the speed up factor of two is roughly the expected baseline that stems from the eradication of memory management overhead as well as from the eradication of copy operations, the further improvements can be attributed to architecture specific effects, such as improved cache behaviour or the improvement of lock-step execution.

The resulting codes match their imperative counterparts, showing how we are closing the performance gap between side-effect-free, data-parallel languages, and imperative, low-level languages.

While this may appear to be a rather specific issue in the context of compiling data-parallel functional programs, we expect further practical applicability in the context of code generation for domain specific languages (DSLs). Whenever basic building blocks of array manipulations are encapsulated in individual functions, which are expected to be combined at later stages to achieve more complex operations, the identification of potential reuses for entire arrays is crucial for achieving good runtime performance. As this paper shows, polyhedral analyses prove to be an excellent vehicle to determine the potential for such in-place updates.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and suggestions.

8. REFERENCES

[1] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[5] D. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989.

[6] A. Cohen. Parallelization via constrained storage mapping optimization. In *Lecture Notes in Computer Science*, pages 1615–83, 1999.

[7] E. De Greef, F. Catthoor, and H. De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Comput.*, 23(12):1811–1837, Dec. 1997.

[8] S. Fitzgerald and R. Oldehoeft. Update-in-place Analysis for True Multidimensional Arrays. In A. Böhm and J. Feo, editors, *High Performance Functional Computing*, pages 105–118, 1995.

[9] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller. Compiler optimizations for scalable parallel systems. chapter The Sisal Project: Real World Functional Programming, pages 45–72. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[10] C. Grelck and K. Trojahner. Implicit Memory Management for SaC. In C. Grelck and F. Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, pages 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics, 2004. Technical Report 0408.

[11] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ACM International Conference on Supercomputing (ICS'13)*, Eugene, OR, June 2013. ACM Press.

[12] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *POPL '85*, pages 300–313. ACM Press, 1985.

[13] V. Lefebvre and P. Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In *In Proc. EuroPar Conference*, pages 356–363. Springer Verlag, 1997.

[14] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, Mar. 1970.

[15] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, Sept. 2000.

[16] S. B. Scholz. Single Assignment C – Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[17] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, San Jose, California, October 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[18] A. Šinkarovs, S. Scholz, R. Bernecky, R. Douma, and C. Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience*, 2013.

[19] V. Wieser, C. Grelck, P. Haslinger, J. Guo, F. Korzeniowski, R. Bernecky, B. Moser, and S. Scholz. Combining high productivity and high performance in image processing using Single Assignment C on multi-core CPUs and many-core GPUs. *Journal of Electronic Imaging*, 21(2), 2012.

[20] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 389–397. Springer Berlin Heidelberg, 1996.