INTERMEDIATE VARIABLE ELIMINATION IN A GLOBAL CONTEXT FOR A 3D MULTIMEDIA APPLICATION

Marc Leeman[*]

ESAT/KULEUVEN,
Kasteelpark Arenberg 10,
B3001 Leuven Belgium

David Atienza Alonso[†]

DACYA/UCM, and IMEC vzw,
Avda. Complutense s/n, Kapeldreef 75,
28040 Madrid Spain 3000 Leuven Belgium

ABSTRACT

The 3D multimedia applications have been experiencing recently a tremendous growth in number and complexity. Such applications mainly consist of complex algorithms that process extensive amounts of data to create 3D images and results. For quick access, data need to be stored in small and expensive memories near the processor. Due to the increasing memory-processor gap in speed and the characteristics of multimedia applications (with highly power- and space-consuming data sets), software transformations are required to decrease memory requirements. In this paper, we propose a method to reduce the indirections of data types in real 3D multimedia applications. It is based on software transformations of the original algorithm to minimize the intermediate assignments and, as such, the required data types. To assess the performance of our method, we apply it to a relatively new 3D image reconstruction application. As a result, for this multimedia application, our method reduces 50× the amount of memory accesses, 30× the normalized memory footprint and 67× the energy consumption compared to a manually well-optimized version of the algorithm. Finally, compared to the original application, the overall performance improves by 40% on a PC.

1. INTRODUCTION

The complexity of 3D multimedia applications and platforms has enormously increased in the last years. These applications currently process large amounts of data and require a good level of performance. Representative examples where they are used can be archaeological site recording and reconstruction, architectural planning, augmented reality and film industry [5, 11, 9]. In all these examples, this kind of applications has to be ported to handheld visualization devices to achieve quick on-site visualization and processing. Consequently, they need optimal algorithms and efficient techniques to access and place the data in fast registers or memories. However, an important part of the data will remain in main memory and it is very well known that the gap between the speeds of this main memory and the processor is growing. Therefore, a combined hardware-software solution is required, i.e. fast processing and storage components and optimal transformations in the source code. This should allow to conveniently use dynamic data structures and minimize the intermediate variable assignments. These code transformations should be done in a systematic way to enable their application in the development of any multimedia application.

In this paper, we assess in a relatively new 3D image reconstruction algorithm [8], the performance of our approach to minimize intermediate assignments and the dynamic data memory used. The results demonstrate a huge improvement in memory footprint, memory accesses, estimated energy dissipation and global performance compared to manually optimized implementations. We have verified that our approach is also applicable to other multimedia applications.

2. RELATED WORK

Recently research has started to address optimal organizations of dynamic DTs and techniques to access them [14]. New system-level approaches for general-purpose design that reduce the power consumption are explained in [3]. Finally, a lot of research has been developed to achieve an optimal dynamic memory management in a general way [13].

In addition, several kinds of transformations to simplify local loops in imperative programs have been considered important techniques in compilers for a long time [7]. However,they often limit themselves to simple Data Types (DTs). An extension to arrays is presented in [7] and a number of dependence tests for array accesses is given in [7]. Moreover, an additional extension to analyze constant dependencies with conditional branches is explained in [12]. Nevertheless, none of these techniques are suitable for the complex dynamic data structures used in multimedia applications because they handle only simple DTs. They are not able to analyze dependencies between loops inside complex data flows as in typical multimedia applications, with many different functions and dynamic DTs involved.

3. APPLICATION DESCRIPTION

Modeling of 3D objects from image sequences is one of the challenging problems in computer vision and has been a research topic for many years. The 3D image reconstruction algorithm where we have applied this method is a recent approach that extracts complex 3D scene models from images. It combines state-of-the-art algorithms for uncalibrated projective reconstruction, self-calibration and dense correspondence matching [8]. Furthermore, it is heavily characterized by intensive internal dynamic memory use to process the different input images.

Apart from multiple frames, this algorithm requires no other information to create a 3D scene. This makes the code especially useful for situations where extensive 3D setup with sensitive equipment is difficult (e.g. crowded streets or

[*]Additional Authors: G. Deconinck, V. De Florio and R. Lauwereins. This work is partially supported by the Fund for Scientific Research - Flanders (Belgium, F.W.O.) through project G.0036.99 and a Postdoctoral Fellowship for Geert Deconinck.

[†]Additional Authors: F. Catthoor (at IMEC) and J. M. Mendias (at DACYA/UCM). This work is partially founded by the Spanish Government Research Grant TIC2002/0750 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031.

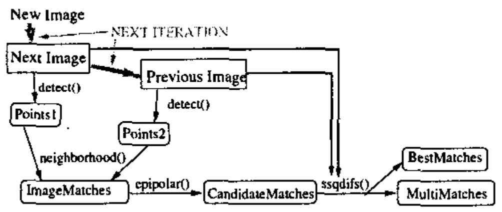

Fig. 1. Dataflow in the corner matching algorithm with the dynamic data types involved

remote locations) or impossible (e.g. the scene is no longer available). Current examples include photo-realistic measurement and reconstruction of ancient lost worlds [5] or 3D modeling of architectural heritage [11]. In these examples, for quick on-site visualization and real-time processing of more frames (for a more detailed reconstruction), speeding up the application is very important and demands extensive code transformations and optimizations. Moreover, for handheld visualization devices, energy consumption is very relevant.

The software module used in the projective reconstruction of this algorithm is one of the basic building blocks in many current 3D vision algorithms: *feature selection and matching*. Since films or pictures are a projection from 3D to 2D, without considerable assumptions on the scene, it is impossible to regenerate the lost information with a single image. With multiple images, the relation between two or more can be used to reconstruct the 3^{rd} dimension.

In this algorithm, the reconstruction is done in a stepwise process, where information is gradually retrieved, evaluated and refined. In a first phase, corners are detected and related between different images. Next, information about the position of the camera is reconstructed and used to find more matching corners. Finally, the matches are used to regenerate a 3D scene. In Figure 1, this corner matching process is depicted for one iteration. In the first iteration, two new input frames are sequentially processed and their corners are detected [6]. Points in each others' neighbourhood or along each others' epipolar line [8] are stored in `ImageCandidates` and become candidates for the following steps of the algorithm. After another epipolar check [8], they are added to `CandidateMatches`. Then, the sum of square differences in the regions (`ssqdifs()` in Figure 1) around the points are checked [8] and if they are under a certain threshold, they are stored in the data structures `MultiMatches` and `BestMatches`. In the next iteration, a new image is processed and compared with the most recent one from the previous iteration, which is now the previous image in Figure 1. This iterative behavior with more new images can be applied as long as necessary to refine the current result.

The further the reconstruction progresses, the smaller the amount of data used by the operations is, and consequently data can be stored in fast memories. This is not possible in the initial phases, especially the corner matching module demands extensive dynamic memory allocation because the amount of candidates for each input varies enormously based on the image properties and the corner detection step parameters. Therefore, an accurate estimation of the memory usage at compile time is nearly impossible and intermediate dynamic DTs are used for every logical step. These first steps are the most critical ones in terms of memory access optimisation. In fact, many state-of-the-art 3D vision algorithms include this behavior, using some sort of *candidate selection* followed by *a criterion evaluation* in a sequence of logical steps [1].

4. METHOD DESCRIPTION

In many 3D multimedia algorithms, the processing steps are mostly driven by regular iterative operations, i.e. the input data is accessed in loops and inside them a sequence of independent steps or functions are applied. The data is traversed and processed in several places in the code. As a result, profiling information showed that dynamic short-lived intermediate data sets create an additional dynamic memory footprint overhead of 30% to 40% of the total amount of memory needed in the application. The sizes vary from 0.5 to 600 KB and the accesses to the data must use large memories, which are slow and highly power-consuming, especially if they are off-chip.

To solve this, since compilers and hardware are not able to figure out the complex dependencies of multimedia applications on code yet, source code transformations are required. One of the promising techniques to reduce memory access overhead is copy propagation [4]. In a very simple way, the basic idea is that, after an assignment, e.g. `g = f(x)`, if it is possible to use `f(x)` instead of `g`, the assignment of the intermediate variable can be removed. Another relevant technique to increase performance is the selection of an optimal number of loops and their order of execution. In this case, transformations like loop in-lining, loop distribution, loop interchange or loop merging should be considered, as they are currently used to increase temporal and spatial data cache locality [10].

These aforementioned techniques are complementary and should be applied, where they are most effective, at a very high-level of the optimizations in multimedia applications, before any kind of refinement in the memory management. Moreover, for the best results with a global view of the algorithm, these optimizations must be done over function boundaries and across different condition and loop scopes. To this end, in our method selective in-lining is used.

In order to improve the dynamic behavior in the automated 3D reconstruction example, several steps were necessary. The following code fragment (in Figure 2) illustrates how the combination of these techniques is used on a cross function boundary level. `function1` processes all data in `a` and stores it in `b`. The subsequent function processes this `b` again and adds the evaluated data in `c`. Since all data generated by `function1` needs to be processed and evaluated in `function2`, they can be combined when no global data-dependency requires all data in `b` (e.g. random sample generation).

```
function1(a,b)               functionA(a,c))
 foreach x in a               foreach x in a
  if(condition1(f(x)))         if(condition1(f(x)))
    b->add(f(x));                if(condition2(g(f(x)))
                                    c->add(g(f(x)));
function2(b,c)
 foreach y in b
  if(condition2(g(y)))
    c->add(g(y));
```

Fig. 2. Very simple example of transformations

However, global code transformations yield an additional complexity of references and overhead in code size. Hence, clear guidelines must be followed before applying them. First, operations should be moved outside loops (up in the code) as much as possible, without breaking data-dependency constraints. Second, in order to inline a function in a loop, the number of times the function gets exe-

Version	memory access	memory footprint (B)	energy (μJ)
Original	93,489,433	312,713	17,480,062
Static	728,289	216,748	187,813
Optimized	15,486	7,201	2,787

Table 1. Dynamic memory use

Data Types	Original	Optim1	Optim2	Final
Original DTs	205.33	1256.14	99.06	97.90
Optim. dyn. DTs	180.85	98.48	97.25	98.58

Table 2. Running time for 100 image pairs and the stepwise process to optimize the DTs (in secs)

cuted (uses loop information) must be sufficiently high, we have estimated with analytical measures that at least 60% of the overall amount of processing time of the loop (e.g. functions using only one or two elements identified by a loop index should not be inlined). Third, the index function of the control flow (in our example, exhaustive traversal) must be simple. It should not involve complex index composition transformations. Finally, output data can be written unambiguously in function of the input data, i.e. the input data can be traced back knowing the output (injective relationship).

5. METHOD APPLICATION AND RESULTS

In this section, our method is applied to a representation of the core code of the corner matching algorithm (note that these are only relevant pieces to show the applied transformation. For the full code of the entire 3D algorithm, which contains 1.75 million lines of high level C++, see [1]).

The matching phase can be subdivided in logical substeps processed as a pipeline with intermediate variables as buffers. Each of them evaluates input data and saves results.

According to the algorithm, in the original code sample in Figure 3, all the index pairs of points that are in a search region (in a window along the epipolar line [8]) are generated and returned by `GetInSearchRegion()`. Next, the initialization of a section threshold is done followed by a loop over the `corners1` elements. For each of these elements, the corresponding ones are fetched from the `CandidateMatches` by searching in `corners2`. Finally, a sum of squared differences criterion is used to compare neighboring corner candidates and determine if they match [8].

While analyzing this code sample, a number of problems are observed. First, a loop over `corners1` is present twice in the overall code and at the same "depth" (depth increases by entering loop or conditional bodies). Second, in several places, a function returns a list of meta data: the elements are not returned, but their location in a list. Then, this meta data is used again to do an expensive lookup for the values required to construct it previously (`GetInSearchRegion()`, `GetSubList()` and `PointsInRegion()`).

The first thing to do is moving non-loop and conditional code upward (as early as possible) without breaking data-dependencies to remove code redundancy and simplify the control-flow. In the code extract, this is the initialization code that does not use loop indices. It is not directly used by code dependent on loop indices and it is not changed within the loops by variables dependent on loop indices. Consequently, the initialization of `thresh` is moved to the beginning. In addition, `best` is reset and

```
CandidateMatches = \
  GetInSearchRegion(frame1,frame2,corners1,corners2);
thresh=InitThreshold();
for(corner1;corner1<corners1->dimx;corner1++){ // (1)
  best=bestinit;
  L2=GetSubList(CandidateMatches,0,corner1);
  for(corner2=0;L2->dimx;corner2++){            // (2)
    C2=GetElement(L2,corner2);
    if(curtresh=GoodStrength(C2,thresh){        // (a)
      MultiMatches->add([corner1,corner2]);
      if(curtresh<best){
        best=corner2;
  } } }
  if(best){
    LocalMatches->add([corner1,best]);
} }

TL GetInSearchRegion(image1,image2,point1,point2){
  for(i=0;i<point1->dimx;i++){                 // (3)
    Corner1=GetElement(point1,i);
    ImageMatches = \
      PointsInRegion(Corner1,Window,points2);
    for(j=0;j<ImageMatches->dimx;j++){         // (4)
      Corner2=GetElement(ImageMatches,j);
      if(epipolar(Corner2)){                   // (b)
        result->add([i,j]);
  } } }
  return resultL;
}

TL PointsInRegion(Point, Window, List){
  for(i=0;i<List->dimx;i++){                   // (5)
    Point2=GetElement(List,i);
    if(IsInWindow(Window,Point2)){             // (c)
      List->add(Point2);
  } }
  return List;
}

TL GetSubList(List,n,eval){
  for(i=0;i<List->dimx;i++){                   // (6)
    Match=GetElement(List,i);
    if(Match[n] == eval){                      // (d)
      RList->add(Match);
    }}
    return RList;
}
```

Fig. 3. Code sample from the corner matching step

used by loop dependent data and cannot be moved outside the loop. Secondly, `GetSearchRegion()` is inlined and since no data-dependency problems exist, loops labeled as 1 and 3 in Figure 3 are merged. Moreover, since `PointsInRegion()` accesses a lot of elements, it is inlined as well. The result of `PointsInRegion()`, the data structure `ImageMatches`, is used again to do additional epipolar checks, labeled as `(b)` in Figure 3. However, the data in the evaluation does not require order or other properties and this check is more efficiently done after creating the elements for `ImageMatches` or the condition `(c)` in Figure 3.

In addition, `ImageMatches` is no longer used and can be eliminated by advanced signal substitution [4]. At this point, a data bottleneck between the loops 5 and 2 exists: if `GetSubList` requires all the data of `CandidateMatches`, no further elimination can be done. `GetSubList` returns subsets of `CandidateMatches`, those involving the `corner1` in the merged loop 1 and 3. Then, loop 2 is a subset of loop 5. Also, the sum of loop 2 iterations match those of loop 5. As a result, `GetSubList` returns all the elements in `CandidateMatches` without overlapping. Likewise, this process is done for `CandidateMatches` and `GetInSearchRegion()`, and `CandidateMatches` is removed.

Finally, the code in Figure 5 is obtained. Only 2 loops over the main corner lists and the conditions are retained. Note that not all the code is inlined at this level. The three function calls that implement the conditions remain unchanged (they operate only on a couple of index dependent

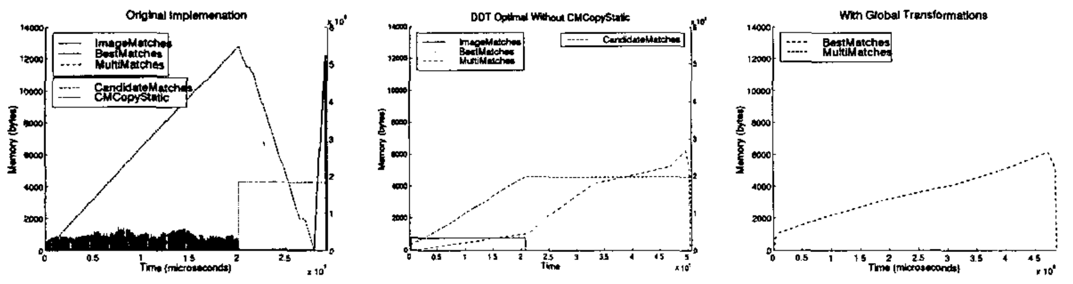

Fig. 4. Memory footprint over time. All plots are mapped on the left axis, but `CandidateMatches` and `CMCopyStatic` (right axis). Left, the original implementation, center, optimal implementation with dynamic memory without applying our method, right after it.

elements) and the code size is not unnecessarily increased. On the contrary, the overall size improves due to intermediate variable elimination by 10%.

```
thresh=InitThreshold();
for(i=0;i<corners1->dimx;i++){                      // (1)
  best=bestinit;
  Corner1=GetCorner(points1,i);
  for(j=0;j<corners2->dimx;j++){                    // (2)
    Corner2=GetCorner(points2,j);
    if(IsInWindow(window,Corner2)){                 // (c)
      if(epipolar(Corner2){                         // (b)
        if(curthresh = \
             GoodStrength(Corner1, Corner2){ // (a)
          MultiMatches->add([corner1,corner2]);
          if(curtresh<best){
            best=corner2;
      } } }
      if(best){
        LocalMatches->add([corner1,best]);
} } } }
```

Fig. 5. Final code sample for the corner matching step

The overall effects on memory usage and accesses to the intermediate dynamic data are shown in Table 1. The static solution numbers are obtained by replacing all the dynamic DTs with static arrays, and only substituting the costly `GetSubList()`. Though the improvement is significant, a quick comparison shows that a global optimization with our method achieves much better results. It improves the figures by almost 50× for memory accesses, 30× for normalized memory footprint and 67× the energy consumption (using the model [2]) compared to an optimized static version.

Also, the runtime to process 100 images [1] is shown in Table 2. It shows a stepwise application of our approach. The first row uses the original DTs, double linked lists, while the second one uses optimized dynamic ones [14]. Although the final figures converge, Table 2 shows that a partial application of our method can have negative performance (cycle count) results sometimes. It should be extensively applied to obtain the full gain. Note also that the runtime for the optimized static solution (not shown in the figure) is slightly better (95.42 s). However, the flexibility to process the input images is removed by fixing the possible memory use and the program will fail with extreme values of the parameters of the system and cause deadline misses.

After the transformations, automatically generated profiling information was obtained for relevant versions of our example and they are shown in Figure 4. As we show in the code, first `ImageMatches` and `CandidateMatches` are created. These are followed by `CMCopyStatic` and finally `Multimatches` and `BestMatches`. As it is shown in Figure 4 and Table 1, our final optimized version reduces 30× the dynamic memory footprint, 67× the estimated consumed energy and 50× the memory accesses.

[1]The results were all obtained with `-O2`, gcc 3.2.2 on an AMD650 with 512 MB SDRAM and running GNU/Linux 2.4.20

6. CONCLUSIONS

The complexity of the 3D multimedia applications has enormously increased and they process extensive data to create 3D images and results. Therefore, they need complex DTs and software transformations to minimize their indirections. In this paper, we have demonstrated on a relatively new automatic 3D image reconstruction algorithm the efficiency and necessity to perform profound global transformations, i.e. loop transformations, selective inlining and advanced signal substitution. In fact, where dynamic memory is used for temporary intermediate values, these techniques are able to reduce or remove intermediate assignments and data buffers. Compared to conventionally optimized versions, a global program acceleration is accomplished and significant improvements in memory footprint, estimated energy dissipation and global performance for typical multimedia applications as well. Moreover, we have characterized our code transformations to show its applicability to other 3D multimedia applications.

7. REFERENCES

[1] Target jr, 2002. `http://www.targetjr.org`.

[2] B. S. Amrutur and M. A. Horowitz. Speed and Power Scaling of SRAM's. *IEEE Trans. Solid-State Circuits*, 35(2), Feb, 2000.

[3] L. Benini and G. De Micheli. System level power optimization techniques and tools. In *ACM TODAES*, April 2000.

[4] F. Catthoor et al. System-level data-flow transformation exploration *J. of VLSI Signal Processing, Kluwer*, Vol.18(No.1):39–50, 1998.

[5] J. Cosmas et al. 3D murale, 2002. `http://www.brunel.ac.uk/project/murale/home.html`.

[6] C. Harris and M. Stephens. A combined corner and edge detector. In *4th Alvey Vision Conference*, pages 147–151, Manchester, 1988.

[7] S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publisher, San Francisco, CA, 1997.

[8] M. Pollefeys. *Self-Calibration and metric 3D reconstruction from uncalibrated image sequences*. PhD thesis, K.U.Leuven, May 1999.

[9] R. Rowe. Industrial light and magic. *Linux Journal*, July 2002.

[10] U.Banerjee et al. Automatic program parallelisation. In *Invited paper, IEEE Int. Conference*, volume 81, pp. 211–243, Feb 1993.

[11] M. Pollefeys. 3D Modelling Projects - M. P. Home page, 2002. `http://www.esat.kuleuven.ac.be/~pollefey`.

[12] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Prog. Lang. and Systems*, 13:181–210, 1991.

[13] P. Wilson et al.. Dynamic Storage Allocation, A survey and critical review. *Int. Workshop on Mem. Manag.*, Scotland, UK, 1995.

[14] Ch. Ykman-Couvreur et al., Dynamic Memory Management Methodology Applied to Embedded Telecom Networks. *Trans. VLSI*, Oct, 2002.