System-Level Memory Management for Weakly Parallel Image Processing*

Koen Danckaert[1], Francky Catthoor[2] and Hugo De Man[2]

[1] IMEC, VSDM Division, Kapeldreef 75, B-3001 Leuven, Belgium
[2] IMEC, VSDM division and Katholieke Universiteit Leuven

Abstract. Application studies in the domain of image and video processing indicate that between 50 and 80% of the area cost in (application-specific) architectures for multi-dimensional (M-D) signal processing is due to *memory units*. This is true for both single-processor and weakly parallel processor realizations. This paper has two main contributions. First, to reduce this dominant cost, we propose to address the *system-level storage organization* for the M-D signals as a first step in the overall methodology to map these applications. Secondly, we will demonstrate the usefulness of this novel approach based on a realistic image processing test-vehicle, namely a cavity detection algorithm. The novel design results for this relevant application are useful as such.

1 Introduction and Related Work

In multi-media applications and others that make use of large multi-dimensional data structures, a considerable amount of memory is required. The ever increasing storage requirements in these applications make the memory cost usually one of the dominant contributions to the total system cost. This is especially true for embedded systems [8, 12].

Up to now, only few hardware synthesis systems [12, 16, 18] try to reduce the storage requirements for array-type data structures, always focussed on single-processor realizations and with (severe) model limitations. Also in our own previous work on ATOMIUM [14, 19], we have only focussed on single-processor storage, dealing with loop transformations, memory allocation and in-place storage reduction for complex M-D signal processing.

Although many software compilers try to come up with the best array layout in memory for optimal cache performance (see e.g. [6, 9]) they do not try to directly reduce the storage requirements as memory is allocated based on the available variable declarations. However, in general, this can lead to a large over-allocation, compared to the maximal amount of memory which is really needed over time. Moreover, the number of transfers to large memories is not fully minimized this way. Consequently, there is a loss in power consumption and overhead cycles.

* This research was partly sponsored by the JESSI AC75 project of the EC

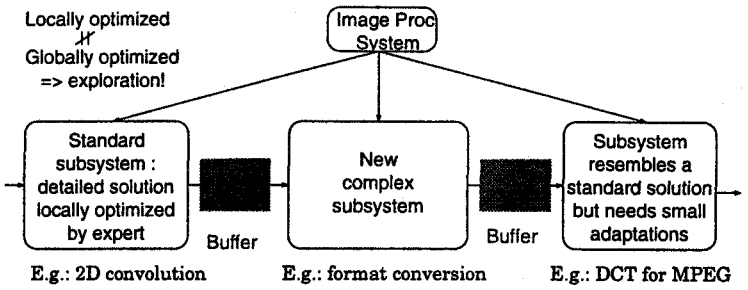

Fig. 1. Data transfer and storage exploration for data dominated systems

To remedy this situation, the system-level memory management (SLMM) oriented methodology presented in this paper applies more aggressive loop and data flow transformations than previously done. These transformations are able to significantly reduce the storage requirements for statically allocated memory. Moreover, the previously applied compiler techniques tackle the parallelization and load balancing issues as the only key point so they perform these first in the overall methodology [1, 2, 4, 5, 13, 15, 20]. Typically, also regular loop scheduling [7] or irregular multi-processor scheduling [10, 11, 17] techniques are performed during or just after this stage. For the typical image processing system in Fig. 1, this means that all the subsystems are treated separately and each of these will be compiled in a locally optimized way onto the parallel processor. This strategy leads to a good parallel solution but unfortunately, it will typically give rise to a significant buffer overhead for the mismatch between the data produced and consumed in the different subsystems. To solve this, our approach first applies storage and transfer oriented optimizations *between* the different systems. Initially, all the subsystems containing M-D processing are combined into one global specification model, and then optimized as a whole in terms of SLMM, prior to the other tasks.

2 Problem Definition

As indicated, in our approach we will first perform the storage and transfer optimization, before other optimizations or parallel processor mapping stages. The input will be a control/data flow-graph CDFG model where all data dependencies have been made explicit to allow formal compile-time analysis, i.e. corresponding to a single-assignment description.

We also require that the subsystems are not treated separately but as a whole. For the typical image processing system in Fig. 1, this means that initially, all the subsystems containing M-D processing are combined into one CDFG "bubble", specified in single assignment form wherever possible and then optimized as a whole in terms of SLMM.

The SLMM stages internally can then be applied in terms of loop/data flow transformations, memory hierarchy and allocation decisions, and in-place map-

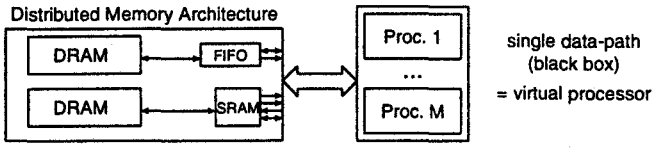

Fig. 2. Architectural model for SLMM

ping, largely similar to the single-processor case [14]. These stages treat the data-path as if it is one global data-path (Fig. 2).

The output of the SLMM stage will be a transformed optimized CDFG with restrictions on control and data flow, and a preliminary distributed background memory organization. These two descriptions can then be used as input for the subsequent processor partitioning and load balancing stage and from there to the final processor HW synthesis or SW compilation stages, dealt with by more traditional techniques.

If possible, the partitioning has to be done in a way that is not cutting over process communication links which involve M-D streams produced and consumed differently — so only scalar streams[3] are cut. If this is not feasible, the two communicating subprocesses have to be connected by a memory subsystem which is shared between the corresponding processors.

More details and formalization of the tasks in this methodology will be discussed in future papers. Here, due to lack of space we will only illustrate our approach in an intuitive way on a typical test-vehicle.

3 Illustration on a Realistic Design: Cavity Detection

Cavity (or edge) detection is an important step in many (in particular medical) image processing applications [3]. This algorithm mainly consists of a sequence of four distinct steps, each of which computes new matrix information from the output of the previous step. So, these steps can be seen as separate processes. A classical approach to parallelize this irregular algorithm on a weakly parallel processor [10, 11, 17] would be to assign each of these steps to its own processor, and to pipeline the processing in a coarse-scale way: while processor 1 is working on frame x, processor 2 is working on frame x-1, and so on. This means, however, that we need an enormous amount of memory: one image frame per processor, plus one for reading in a new frame (as in Fig. 1). All these frames have to be stored in background memory, and have to be transferred between a processor and a memory in each step. So this is unacceptable from the viewpoint of both storage area and overall power.

The proposed SLMM approach can significantly reduce the area and power requirements of this application. To illustrate this, we will first consider the

[3] Note that a scalar signal could also be an initially M-D signal which has been "projected" into a sequential scalar communication

```
function GaussBlur (image_in : W[N][N]) image_out : W[N][N] =
  begin   //Compute horizontal weighted average
    (x : 0 .. N-1):: (y : 0 .. N-1):: begin
      tmp1[x][y][0]=0;   (k : -GB .. GB)::
        tmp1[x][y][k+GB+1] = tmp1[x][y][k+GB] + Gauss[k]*image_in[x+k][y];
    end;
    { Compute vertical weighted average (analogous) }
  end;
function ComputeEdges (image_in : W[N][N]) image_out : W[N][N] =
  { Replace every pixel with the maximum of difference with its neighbors }
function Reverse (image_in : W[N][N]) image_out : W[N][N] =
  { Search for the maximum value that occurs : maxval }
  { Subtract every pixelvalue from this maximum value }
function DetectRoots (image_in : W[N][N]) image_out : bool[N][N] =
  { image_out[x][y] is true if no neighbors are bigger than image_in[x][y] }
function LabelRoots (image_in : bool[N][N]) image_out : int[N][N] =
  { Analogous, but only looks at neighbors on line y-1 (not y+1). }
```

Fig. 3. Initial description of the cavity detection algorithm

mapping of the algorithm on one virtual processor, as illustrated in Fig. 2. Then we will look at some ways to parallelize the result without sacrificing storage and transfer overhead for improved throughput or good load balancing.

The main functionality of the algorithm as specified by the designers is given in Fig. 3. The four steps are the functions GaussBlur, ComputeEdges, DetectRoots and LabelRoots. There is another function, Reverse, which is executed between ComputeEdges and DetectRoots, but we will show that this function can be removed as part of a system-level data-flow transformation. We will assume that the image enters and leaves the system in the conventional row-wise fashion. Also other formats exhibit similar optimization possibilities however.

3.1 Optimized Memory Organization on One Virtual Processor

In the initial description of the algorithm, each step is applied to the image as a whole. From the viewpoint of memory management, this is a bad solution, as it means that in every step, the whole image must be read from background memory and written back to it. A better solution would be to read one row of the image, and apply all steps of the algorithm to this row directly. However, this is not possible as such: for example, to apply ComputeEdges to line y, GaussBlur must already have been applied to line y+1. Furthermore, some local transformations are needed. These issues will be discussed next.

Local Optimization in GaussBlur. In the first part of GaussBlur, every pixel is replaced by a weighted average of its horizontal neighbors. In the initial

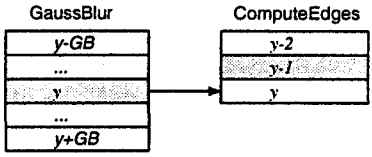

Fig. 4. Shaded boxes indicate computations.

description, this is executed in a column-wise fashion. So, for every pixel, all its horizontal neighbors have to be read from background memory, *or* we need a foreground buffer of 2*GB+1 columns (GB is typically 1 or 2). Moreover, because the image enters the system row-wise, we need a background memory in which the next incoming frame can be stored while we are processing the previous one. If we interchange the x and y loops, there is much better data locality: we only need a foreground buffer of 2*GB+1 pixels. Also we don't need an extra background memory, as the processor can now accept the rows of the image as they enter the system.

Now consider the second part of GaussBlur, where a weighted average of the vertical neighbors of every pixel is computed. We can optimize memory usage in the same manner as above, in which case we have to work in a column-wise fashion this time. As the output of the previous step is row-wise, this means that we have to write the whole image to background memory between the two steps. So, here it will be better to continue to work row-wise, which means we only need a foreground buffer of 2*GB+1 rows.

Similar Local Transformations can be applied to the other functions. For example, in ComputeEdges, we only need the eight direct neighbors of every pixel to compute its new value. So to compute the values of line y, we need the pixels of lines y-1, y and y+1. This means that a buffer of three lines is sufficient to avoid extensive data transfers from background memory to data-path. In the second column of Table 1, the results of these local transformations are shown.

Global Transformation of GaussBlur and ComputeEdges. To perform ComputeEdges on line y, we need the pixels of lines y-1, y and y+1. So instead of first performing GaussBlur on the whole image, ComputeEdges can be applied to line y-1 already when GaussBlur has just been applied to line y. In this way, we don't need to store the intermediate image in a background memory between these two functions — at least if we use three extra foreground line buffers, as depicted in Fig. 4.

To accomplish this, we have to join the functions GaussBlur and ComputeEdges. Then we can apply a combined loop folding and merging transformation to the y-loop. A similar global transformation can be applied to DetectRoots and LabelRoots, but not to Reverse, as explained in the next paragraph. The

Table 1. Overview of the reduction of background memories and data transfers, accomplished by the different system-level transformations, and the amount of foreground buffers needed. BG=Background, FG=Foreground, 1 BG memory means 1 BG memory for a whole image frame.

	Initial algo	After local trafo	y-loop folded	After data-flow trafo	x-loop folded
BG reads ($*N^2$)	4*GB+26	6	1	0	0
BG writes ($*N^2$)	5	4	1	0	0
FG line buffers	0	2*GB+1	2*GB+4	2*GB+9	2*GB+5
BG memories	3	1	1	0	0

reduction of the number of data transfers as a result of these global transformations, is shown in the third column of Table 1.

The Function Reverse. The technique described above cannot be applied to Reverse, because in this function the maximum value that occurs in the whole image is first computed. If we only consider the number of operations to be performed, the computation of the maximum represents only a small fraction of the total arithmetic effort of the algorithm. However, if we look at the number of transfers and the amount of background memory needed, this computation means that the whole image has to pass through the processor, before the next step of the algorithm can be performed. So we have a huge amount of extra transfers, and we need an extra background memory for one image. From the viewpoint of memory management, the computation of a maximum is a real bottle-neck, that cannot be directly circumvented.

In this case however, the function Reverse is a direct translation from an original system-level description of the algorithm, where specific functions have been reused. It can be avoided by adapting the next step of the algorithm (DetectRoots) by means of a data flow transformation. Instead of $image\_out[x][y] = if(p > \{q\})$, where p,q are pixel elements produced by Reverse, we can write $image\_out = if(-p < \{-q\})$ or $image\_out = if(c - p < \{c - q\})$, where $c = maxval$ is a constant. So instead of performing the Reverse function and implementing the original DetectRoots, we will omit the Reverse function and implement instead:

{ image_out[x][y] is true if no neighbors are smaller than image_in[x][y] }

In this way the storage and access bottle-neck is totally avoided.

The Functions DetectRoots and LabelRoots. Because the DetectRoots and LabelRoots steps are similar to ComputeEdges, we can now apply the same technique as above. Assuming that GB=1, we now need a total of 11 line buffers. Clearly, the amount of data transfers has been further reduced. See Table 1.

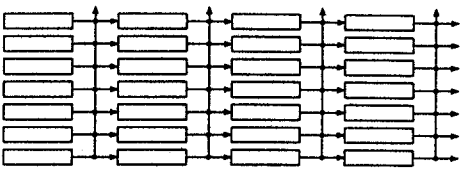

Fig. 5. Flow of data through buffers

Further Optimizations. By applying a loop folding transformation to the x-loop (as we did to the y-loop), another 4 buffers can be optimized away, which brings us to 7 line buffers. See Table 1.

3.2 Parallelization of the storage and transfer optimized solution

In this section we will look at some ways to parallelize the optimized algorithm. We will assume that a speedup of about 4 is required, which corresponds well with the typically available amount of parallelism in the current generation of multi-media processors like the C80 of TI and the TriMedia of Philips.

Irregular Coarse-grain Pipelining. A first method to parallelize the algorithm has already been mentioned in the beginning of this section: coarse-grain pipelining (at the level of the image frames). This can work well for load balancing on a four-processor system, but is clearly an unacceptable method if we have efficient memory management in mind.

Regular Data Parallelism. A second parallelization method (as suggested more by the data partitioning methods supported in [1, 2, 13, 20]) is to distribute the image itself over the four processors. Because the image enters row-wise, we have to choose a column-wise partitioning to keep the processors busy. In this way, we will need 7 line-buffers per processor, or 28 in total, but their length is only a quarter of a line. The flow of data through these 28 buffers is shown in Fig. 5. This is still much more expensive than just 7 simple buffers, which are standard components that can be realized in a very compact way. Moreover idle cycles will have to be introduced with this solution (this can be overcome by replacing the FIFO buffers with SRAM buffers, which are much more expensive).

Fine-grain Pipelining. A third way to parallelize this algorithm consists in assigning each of the steps of the algorithm to a different processor. Still assuming that GB=1, processor one has a buffer of two lines (y-1 and y). Line y+1 enters the processor as a scalar stream; synchronously the GaussBlur-step can be performed on line y, the result of which can be sent to the second processor as a scalar stream. This one can concurrently (and synchronously) apply the ComputeEdges step to line y-1 and so on. In this way we only need a buffer of

two lines per processor (and one for the last processor), or 7 in total! This is the same amount we needed for the mono-processor case. So we have achieved what we were looking for: improved performance without sacrificing storage and transfer overhead (which would translate in area and power overhead).

Note that the load balance will be less optimal than in the data parallel solution. In many cases, the parallelization research community focuses on avoiding idle cycles and achieving a better load balance. For data dominated designs (especially embedded ones) it is however at least as important to look at the data storage and transfer organization. If we can avoid a buffer of 32 Kbit by using an extra processor, this can be advantageous even is this processor would be idle 90% of the time (which would also mean we have a very bad load balance), because the cost of this extra processor in terms of area and power is usually less than the cost of a 32Kbit on-chip memory.

4 Conclusion

In this paper, a summary has been made of the main storage related issues to be resolved in the system-level mapping context of data-dominated signal processing applications on weakly parallel processors. It has been motivated why the novel SLMM methodology proposed here has a good chance to solve many of the efficiency problems which arise when mapping storage dominated image and video processing applications to embedded parallel processors. The feasibility of this approach has been substantiated on a realistic image processing application. Moreover, the requirements on the context of this SLMM work have been addressed, with emphasis on the interaction with the other system-level issues. Several aspects about the internals of the SLMM tool-box are still unsolved but these are a topic of ongoing research.

References

1. S.Amarasinghe, J.Anderson, M.Lam, and C.Tseng, "The SUIF compiler for scalable parallel machines", in *Proc. of the 7th SIAM Conf. on Parallel Proc. for Scientific Computing*, 1995.
2. U.Banerjee, R.Eigenmann, A.Nicolau, D.Padua, "Automatic program parallelisation", *Proc. of the IEEE*, invited paper, Vol.81, No.2, Feb. 1993.
3. M.Bister, Y.Taeymans, J.Cornelis, "Automatic Segmentation of Cardiac MR Images", *Computers in Cardiology*, IEEE Computer Society Press, pp.215-218, 1989.
4. T-S.Chen, J-P.Sheu, "Communication-free data allocation techniques for parallelizing compilers on multicomputers", *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, No.9, pp.924-938, Sep.1994.
5. Y-Y.Chen, Y-C.Hsu, C-T.King, "MULTIPAR: behavioral partition for synthesizing multiprocessor architectures", *IEEE Trans. on VLSI Systems*, Vol.2, No.1, pp.21–32, March 1994.
6. M.Cierniak, W.Li, "Unifying Data and Control Transformations for Distributed Shared-Memory Machines", *Proc. of the SIGPLAN'95 Conf. on Programming Language Design and Implementation*, La Jolla, pp.205-217, Feb. 1995.

7. A.Darte, T.Risset, Y.Robert, "Loop nest scheduling and transformations", in *Environments and Tools for Parallel Scientific Computing*, J.J.Dongarra et al. (eds.), Advances in Parallel Computing 6, North Holland, Amsterdam, pp.309-332, 1993.
8. H.De Man, F.Catthoor, G.Goossens, J.Vanhoof, J.Van Meerbergen, S.Note, J.Huisken, "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proc. of the IEEE*, special issue on "The future of computer-aided Design", Vol.78, No.2, pp.319-335, Feb. 1990.
9. C.Eisenbeis, W.Jalby, D.Windheiser, F.Bodin, "A Strategy for Array Management in Local Memory", *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
10. K.Konstantinides, R.Kaneshiro, J.Tani, "Task allocation and scheduling models for multi-processor digital signal processing", *IEEE Trans. on Acoustics, Speech and Signal Processing*, Vol.ASSP-38, No.12, pp.2151-2161, Dec. 1990.
11. D.Lilja, "The impact of parallel loop scheduling strategies on prefetching in a shared memory multi-processor", *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, No.6, pp.573-584, June 1994.
12. P.Lippens, J.van Meerbergen, W.Verhaegh, A.van der Werf, "Allocation of multi-port memories for hierarchical data streams", *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, Nov. 1993.
13. K.McKinley, M.Hall, T.Harvey, K.Kennedy, N.McIntosh, J.Oldham, M.Paleczny, and G.Roth, "Experiences using the ParaScope editor: an interactive parallel programming tool", in *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, USA, May 1993.
14. L.Nachtergaele, F.Catthoor, F.Balasa, F.Franssen, E.De Greef, H.Samsom, H.De Man, "Optimisation of memory organisation and hierarchy for decreased size and power in video and image processing systems", *Proc. Intnl. Workshop on Memory Technology, Design and Testing*, San Jose CA, pp.82-87, Aug. 1995.
15. C.Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on the architecture design", *IEEE Trans. on Computers*, Vol.37, No.8, pp.991-1004, Aug. 1988.
16. L.Ramachandran, D.Gajski, V.Chaiyakul, "An algorithm for array variable clustering", *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, Paris, France, pp.262-266, Feb. 1994.
17. M.Schwiegershausen, M.Schönfeld and P.Pirsch, "Mapping complex image processing algorithms onto heterogeneous multi-processors regarding architecture dependent performance parameters", *Intnl. Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994. Also in "Algorithms and Parallel VLSI Architectures III" (eds. M.Moonen, F.Catthoor), Elsevier, 1995.
18. J.Vanhoof, I.Bolsens, H.De Man, "Compiling multi-dimensional data streams into distributed DSP ASIC memory", *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pp.272-275, Nov. 1991.
19. M.van Swaaij, F.Franssen, F.Catthoor, H.De Man, "Automating high-level control flow transformations for DSP memory management", *Proc. IEEE workshop on VLSI signal processing*, Napa Valley CA, Oct. 1992. Also in *VLSI Signal Processing V*, K.Yao, R.Jain, W.Przytula (eds.), IEEE Press, New York, pp.397-406, 1992.
20. M.Wolfe, U.Banerjee, "Data Dependence and its Application to Parallel Processing", *Int. J. of Parallel Programming*, Vol. 16, No. 2, pp.137-178, 1987.