SYNTHESIS OF ARRAY ARCHITECTURES FOR BLOCK MATCHING MOTION ESTIMATION: DESIGN EXPLORATION USING THE TOOL **DG2VHDL**

John Bonk *Andrew Stone* *Elias S. Manolakos*

Communications and Digital signal Processing (CDSP)
Center for Research and Graduate Studies
Electrical and Computer Engineering Department
Northeastern University, Boston, MA 02115

ABSTRACT

In this paper we present a design case study using DG2VHDL, a tool which bridges the gap between an abstract graphical description of a DSP algorithm and its concrete hardware description language (HDL) representation. DG2VHDL automatically translates a Dependence Graph (DG) [1] into a synthesizable, behavioral VHDL entity that can be input to industrial strength behavioral compilers for producing silicon implementations of the algorithm (FPGAs, ASICs). Full Search Block Matching Motion Estimation was selected for its current applications (MPEG, HDTV, Video Conferencing) as well as for the richness of literature and architectural exploration over the last decade. We will not only demonstrate here that the behavioral VHDL code produced automatically by the tool leads, after behavioral synthesis, to an efficient distributed memory and control modular array architecture, but will also provide comparative statistics for several new FS-BMA architectures derived for real-time motion estimation.

1. INTRODUCTION

Block Matching Algorithms (BMA) for Motion Estimation are used in many applications in Digital Image Processing. These algorithms are used to encode video frames in order to reduce the amount of video data that needs to be stored or transmitted. The assumption in this type of encoding is that with moving objects, small groups of pels move in the same direction. Motion Vectors are determined by starting with the first frame of a video sequence, and then comparing blocks of pels of this frame with the next frame. Each block of pels in the next (reference) frame is compared with displaced blocks in the previous (search) frame. This is repeated for all blocks contained in the reference frame. The displacement vectors for the blocks are stored or transmitted instead of the pel data. An example Reference Block and Search space are depicted in Figure 1. The shaded region in the Search space corresponds to a displacement vector of (0,0).

The Full Search BMA is the simplest search algorithm, but is also the most computationally intensive BMA. However, the FS-BMA is a very regular algorithm. The 4-D algorithm from [2] is :

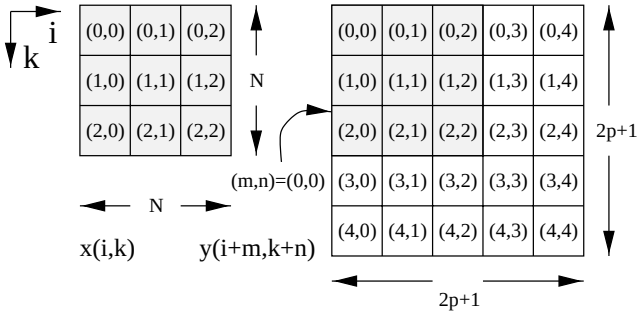

Figure 1: Reference Block and Search Space for $N = 3, p = 1$.

$$s(m,n) = \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} |x(i,k) - y(i+m, k+n)| \qquad (1)$$

$$0 \geq m,n \geq 2p,\ u = min_{m,n}[s(m,n)],\ v = (m,n)|_u \qquad (2)$$

where x is the reference frame, y is the search frame, N is the number of pels in each column and row of the reference block, p is the maximum displacement vector of an $N \times N$ block in each direction within the search frame, u is the minimum displacement metric, and v is the corresponding displacement vector.

Because of the very regular recursive nature of the FS-BMA, the inherent parallelism of the algorithm can be exploited using arrays of processors operating in parallel. In [2], 1-D and 2-D arrays of parallel processors were systematically derived from a 3 -dimensional Dependence Graph (DG). In this paper, we describe three new extensions to the work done in [2] and describe how the exploration and implementation of these designs was greatly facilitated using the DG2VHDL tool.

2. OVERVIEW OF THE DG2VHDL TOOL

DG2VHDL is a tool that tries to facilitate behavioral synthesis by concentrating in the high end of the design flow and in areas where the industrial behavioral compilation

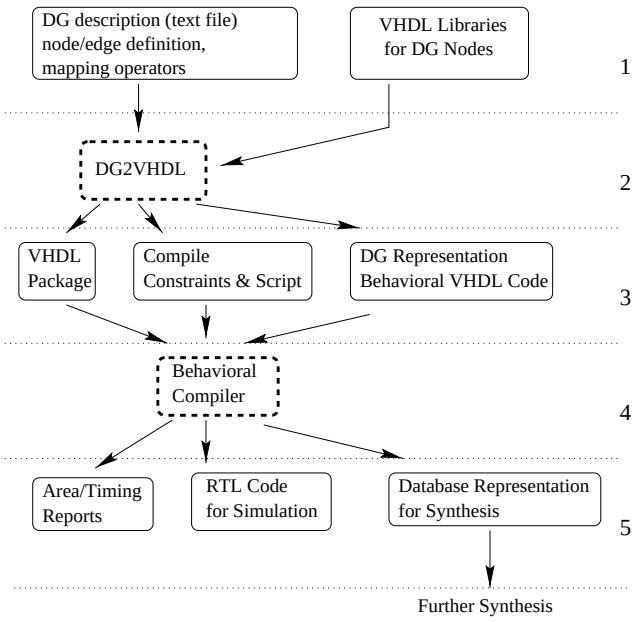

Figure 2: Rapid Prototyping of DSP Algorithms using DG2VHDL.

tools are currently weak, such as: (i) the decomposition of a synthesis problem into smaller ones which can be processed independently; (ii) the explicit expression in the automatically produced behavioral VHDL descriptions of all the available parallelism in the algorithm, so that it gets fully exploited by the behavioral compilation tools, and (iii) the rapid prototyping of high throughput, fully pipelined, easy to modify and resynthesize, modular parallel structures, with distributed memory and control, which can achieve the high throughput rates demanded by real-time signal/image processing applications.

The design process with DG2VHDL can be segmented into several distinct stages, as shown in Figure 2. During phase one, the designer first creates a textual description, or netlist, that describes the nodes and edges of the DG. Every distinct node type should be associated with a behavioral VHDL function in a library. Then the space and time mapping operators should be specified, which will be used to translate the DG into a Signal Flow Graph. The selected mapping may be one that is known to be optimal in some sense (area, efficiency, latency, IO requirements, etc.) or perhaps one the designer merely wishes to explore. The only restriction is that it should be permissible i.e. it does not violate the precedence and space-time compatibility constraints [1]. Mappings that may cause collisions are detected and are not allowed. Finally, the designer should define the bit width of the architecture to be synthesized.

During the second phase of the design process DG2VHDL applies the mapping operators to the input file and generates its own internal representation of the resulting Signal Flow Graph as well as several output files which are representative of the SFG. These include: (i) a VHDL behavioral entity/architecture pair which mirrors the behavior of the SFG and its IO characteristics. (ii) a VHDL package including a declaration of the needed data types and constants, and (iii) third party specific scripts which can be used to drive the hardware synthesis of the SFG using an industrial tool [1].

Upon arriving at phase three, the designer is in possession of several behavioral VHDL files. Simulation should be used to verify their correctness before proceeding with behavioral synthesis. After behavioral synthesis, area and timing reports can be reviewed, and simulation again performed using the produced RTL level VHDL code. If satisfactory, then one may continue onward with gate-level synthesis, targeting a specific silicon implementation (ranging from FPGAs to full custom VLSI devices). At this point the design is simulated again and, assuming an FPGA implementation, it is exported to a particular manufacturer's tool for automated or manual fitting, or perhaps sent to a layout house if a custom layout has been opted for. Apart from hardware testing, the design process is then complete.

For additional information on DG2VHDL, with case study, the interested reader is directed to [4].

3. ALGORITHM DERIVATION

A recursive algorithm for FS-BMA was used to derive a 3-dimensional Dependence Graph (DG) in [2] and is shown in Figure 3. In this 3-D DG (i,k,m), the reference frame $x(i,k)$ is input into the $(i,k,0)$ plane of the DG. The reference data propagates between DG nodes in the [0 0 1] direction. Search frame data, $y(i+m,k+n)$ is input into the $(0,k,m)$ and $(i,k,2p+1)$ planes and propagates in the $[1\ 0\ -1]$ direction. There are three different node types in this DG. The AD nodes compute the Absolute Difference of the two inputs, and adds this value with the difference computed by the node above it. The A nodes add the two inputs together and output the sum. The M nodes output the smaller of the two input values.

This DG (Figure 3) illustrates the comparison of the reference block with $2p+1$ candidate blocks from the search space. For the $m=0$ plane, the reference block is compared with the $m=0$ displacement search block. The difference between the elements of the first column of each block (refer to Fig. 1) is computed along the k direction by the AD nodes in the $(0,k,0)$ direction. The second column difference is computed on the $(1,k,0)$ linear array of M nodes, etc. The column differences are accumulated along the i direction by the A nodes. The last A node output represents the displacement metric for $(0,n)$. This value is then input to the $m=0$ M node which performs the minimum metric comparison and passes the minimum metric on to the next M node. The remaining i,k planes perform similar computations.

In order to map this 3-D DG into a linear Signal Flow Graph (SFG) which is 100% pipelineable, the DG has to be split into three parts (AD nodes, A nodes, M nodes), with each part space mapped independently as described in [5]. This results in a set of three linear arrays. Each array is derived from one set of nodes (AD,A,M). This is similar to architecture AS1 from [5] and [2] with the exception that for this mapping, an x value and y value are input

[1] Currently we target the Synopsys Behavioral Compiler (BC) [3], but other tools are being taken into consideration

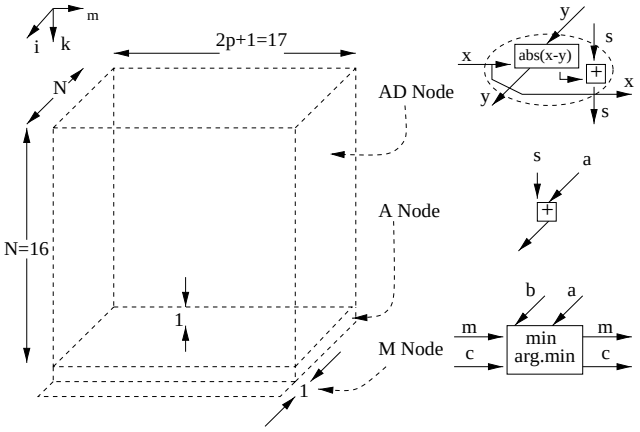

Figure 3: Pirsch 3D DG.

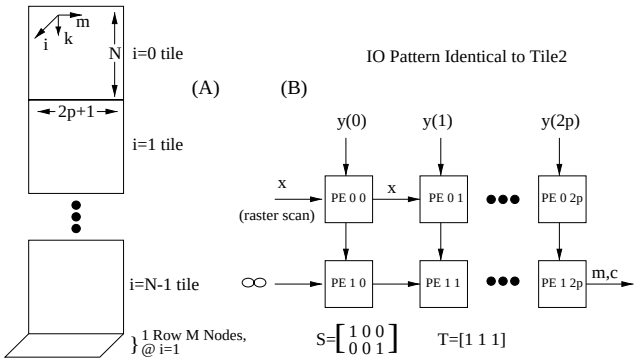

Figure 4: Design Tile 1: (A) DG (B) SFG and IO Patterns.

on each clock cycle, and there are $2p+1$ y values input per clock cycle. The disadvantage of this architecture is that it has $3 \times (2p+1)$ processing elements, and that the linear arrays derived from the AD and A nodes both contain an accumulator. The AD array accumulates the column sum (s edge), and the A array accumulates the sum of the N column sums. The M node array propagates the minimum displacement metric (edge m shown in Fig. 3) along with the arg.min value which is equal to the m plane that produced the minimum result. In order to avoid the duplication of accumulators, we propose a slice and tile variation of the 3-D DG, similar to that used in [6], except that instead of tiling in two directions (vertically and horizontally), the DG is tiled or stacked in only the vertical direction.

Figure 4 (A) illustrates the tiled DG. The tiles are taken from the k, m planes of the original 3D DG from [2], with each plane placed beneath the previous one. The last AD node now produces the final accumulation comparison metric for the entire $N \times N$ block, instead of just for one column as in the original DG, thus no A nodes are needed. By mapping the DG onto the i, m plane, two linear arrays are generated as shown in Fig. 4 (B). This solution again provides the ability to input x and y values on each clock cycle, and has removed the additional accumulation hardware, delay buffers, and the associated control logic.

A second variation of the tiled DG is generated by modifying the Nth AD node to include the M function. Thus the absolute difference, accumulate, and minimum selection functions are all performed by the same node within one clock cycle. This eliminates the extra control needed to synchronize the M function. This DG is shown in Figure 5(A) with the resulting mapping shown in Figure 5(B).

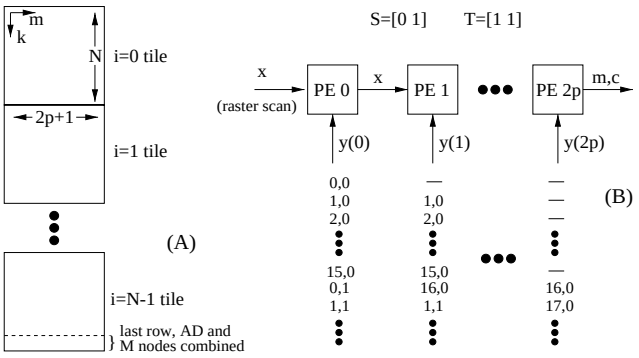

Figure 5: Design Tile 2: (A) DG (B) SFG and IO Patterns.

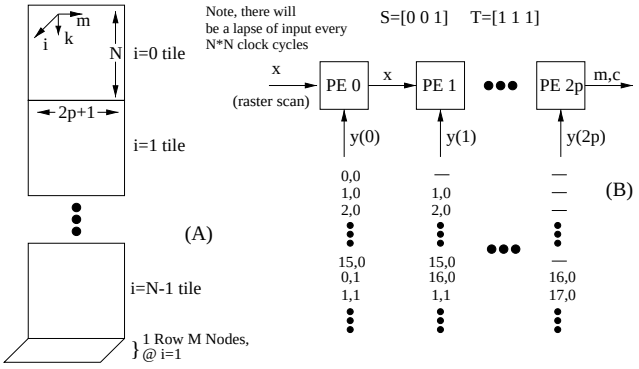

Figure 6: Design Tile 3: (A) DG (B) SFG and IO Patterns.

The third variation uses the same DG developed in Figure 4(A) (also shown in Figure 6) and uses a different space mapping. A space mapping of $S = [0\ 0\ 1]$ was used to produce a linear array of $(2p+1)$ PEs. The resulting SFG is shown in Figure 4(B).

4. SYNTHESIS RESULTS AND COMPARISONS

Each of these three designs was compiled by DG2VHDL and then scheduled and compiled by Synopsys BC Compiler in less than one hour. The target ASIC library used was LSI 10K. Area results are highly dependent on the target library used. Other ASIC libraries may be used which produce outputs with smaller clock periods and less area. Also, an FPGA vendor's library can be used to synthesize designs for FPGA implementation.

Synthesis results and architecture features for the three designs are shown in Table 1. In order to make the results meaningful, the block size was chosen to be 16×16, with the maximum search displacement of +/- 8 (N = 16, p = 8). Also, for x and y inputs of 8 bits, accumulators with 16 bits were used ensuring no overflow errors occur.

Design	Tile1	Tile2	Tile3
Clock Cycle (ns)	36.22	46.07	36.18
Area (gates)	29,026	22,103	25,101
Latency (clock cycles)	273	272	273
Efficiency	0.502	1.000	1.000
Number of PEs	34	17	17
Block Pipeline Period	256	256	257
I/O (pins)	170	170	170

Table 1: Comparative Summary of Designs.

Design Tile 1 has the smallest clock period while maintaining a Block Pipeline Period (β) equal to the problem size (N^2). The disadvantage is that Tile 1 contains twice as many PEs ($2 \times (2p+1)$) as the other two designs, which reduces the overall efficiency of the design and requires extra control hardware which accounts for the higher gate count compared with the other two designs.

Design Tile 2 achieves the smallest total area of the three designs consisting of 17 ($(2p+1)$) PEs. Tile 2 also has a β of N^2 and achieves an overall efficiency of 100%. This means that each PE performs a computation on every clock cycle. The maximum operating frequency of the Tile 2 design is the slowest ($\approx$21 MHz). This is due to the longer execution time needed for the composite AD/M nodes.

Design Tile 3 provides the fastest operating frequency, and contains $(2p+1)$ PEs. It also achieves 100% efficiency, however the Block Pipeline Period is now N^2+1. Thus data can not be input on every clock cycle. There will be one clock cycle out of every N^2 clock cycles when data can not be input to the array.

The I/O for each design is 170 pins. This may be reduced by broadcasting common y input data to multiple PEs and providing an alternate y input port to input data to the PEs which do not receive the common data as was done in [7]. It should be noted that in our implementations the search area is $-8 \leq m, n \leq 8$ compared with the a search space of only $-8 \leq m, n \leq 7$ to in [7]. This results in extra PE(s) in our arrays.

The Tile 1 and Tile 2 designs were synthesized using the Altera Flex 10k FPGA library. Tile 1 and Tile 2 synthesis resulted in projected utilizations of 6873 and 5423 LUTs respectively. Due to limitations in our toolset, the largest device available to us was the EPF10K100 100K gate equivalent FPGA. Both design fit into a two FPGA set (one EPF10K100 device plus one EPF10K50 device). Both of these designs would easily fit in the EPF10K250 device.

Finally, to demonstrate the adherence to the SFG model of the VHDL models targeted by DG2VHDL, we have included a (top level) schematic (Fig. 7).

5. CONCLUSIONS

In this paper we have presented three architectures for Full Search Block Motion Estimation, each providing the designer with the typical tradeoffs of latency and area, but all providing high throughput solutions. These designs were greatly facilitated by the use of DG2VHDL, which provided accurate and rapidly synthesizable VHDL models characteristics of the SFGs shown in Figures 4, 5, and 6. Synthesis from DG2VHDL input file to gate level netlist being performed in less than hour.

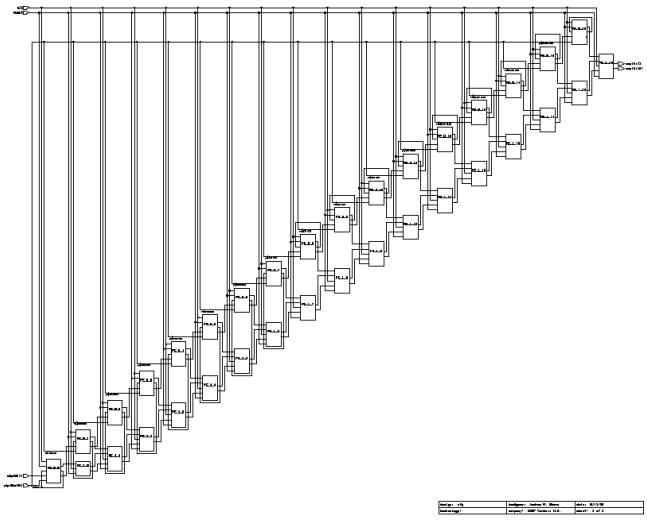

Figure 7: Top level schematic for Design Tile 1 post Synopsys synthesis.

Acknowledgments: The authors would like to thank Synopsys for making their EDA tools available through the University program, and Prof. M. Leeser for permission to use resources in the Rapid Prototyping Lab at Northeastern U.

6. REFERENCES

[1] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, 1988.

[2] T. Komarek and P. Pirsch. Array architectures for block matching algorithms. IEEE Transactions on Circuits and Systems, 36:1301–1308, October 1989.

[3] Synopsys. *Synopsys Online Documentation Collection.*

[4] A. Stone and E.S. Manolakos. Using DG2VHDL to synthesize an FPGA implementation of the 1-D Discrete Wavelet Transform. In Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS), 1998.

[5] P. Pirsch and T. Komaranek. VLSI architectures for block matching algorithms. In SPIE,volume 1001, pp. 882–891, 1988.

[6] S. Chang, J.H. Hwang, and C.W. Jen Scalable Array Architecture Design for Full Search Block Matching IEEE Transactions on Circuits and Systems for Video Technology, 5:332–343, 1995.

[7] C. Sanz, L. de Zubeta, and J. Meneses. FPGA Implementation of the Block-Matching Algorithm for Motion Estimation in Image Coding. 6th Int'l. Workshop on Field-Programmable Logic and Applications, pp. 146–155, Sept. 1996.