IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 8, Issue 7, July 2019

Parallelization for Multi-DSP

Dr. Prakash H T[1], Dr. Srinivas M[2]

Technical Assistant, Department of Technical Education, Board of Technical Examination, Bangaluru, India[1]

Principal and Director Research and Development, St. Mary's Group of Institutions, Hyderabad, India[2]

Abstract: Multi-processor DSPs offer a high-performance cost-effective method that is critical for many embedded application areas. However, it is currently complex and time-consuming to port existing uniprocessor applications to such parallel architectures. There are no commercially available compilers which will automatically map existing sequential DSP programs to a multi-processor machine (Rijpkema et al., 1999). Users are typically required to rewrite their code as a process network or as a set of sequential processes of communication (Lee, 1995). It is well known that such an approach is highly non-trivial and error-prone, possibly leading to a deadlock. This is due to the fact that Digital Signal Processor programs are written using C language rather than FORTRAN language (Hiranandani et al., 1992) and the wide use of pointer arithmetic is shuffled and the compilation of distributed memory space of Digital Signal Processors is difficult. It is a highly specialized skill to rewrite an application in a parallel way. What is needed is a tool that effectively takes existing programs into the new multi-processor architecture and maps them automatically. Although research on auto-parallelising compilation in scientific computing has been going on for over 20 years (Gupta et al., 2000), this has not occurred in the embedded domain. These problems are solved through the use of the pointer conversion technique and a new address resolution technique based on a new data transformation scheme that allows multiple address spaces to be paralleled without the introduction of complex (and potentially deadlocking) passing code message. An auto-parallel C compiler integrating these two techniques into an overall parallel strategy

Keywords: Multi-processors DSPs, Digital Signal Processor programs, Data transformation, Prallelization algorithm, data address resolution

I. INTRODUCTION

Scientific computing domain has been studying auto-parallelizing compilers that take sequential code as input and produce parallel code as output for many years. Multi-processor DSPs are a more recent compilation target within the embedded domain. DSP applications appear to be ideal candidates for auto-parallelization at first glance; many of them have static control-flow and linear access to matrices and vectors. However, due to the widespread practice of using post-increment pointer accesses, auto-parallelizing compilers were not developed (Zivojnovic et al., 1994). In addition, multiprocessor DSPs typically have address spaces that eliminate the need for expensive hardware for memory consistency. This hardware-level saving greatly increases the complexity of the task of the compiler.

```
1. Pointers are Converted to arrays
2. Data Dependence analysis is performed if pointer is free
    a. IF parallel and worthwhile
        i. Determine partitioning of data
        ii. Partition + Data and code transformation
        iii. Resolve the address
3. Generate Parallel code
```

Figure 1. Overall parallelization algorithm

Figure 1 shows the overall parallel algorithm. Analysis of data dependence is achieved by using Pointer to convert the array. Once the program is in a pointer-free form, the standard data-dependence analysis is applied to determine whether the program is parallel and if so, a check is applied to see if the amount available justifies parallelization.

Data parallelism in DSP programs is demoralized by dividing data and computing across processors using a computing owner-computing, SPMD model. For many years, the selection of the best data partition was studied (Lim et al., 1999) and is NP-complete. A simple method of demoralizing parallelism and reducing communication is used in this work; it is also possible to use more advanced partitioning schemes, e.g. affine partitioning (Lim et al., 1999). The key point here is that the approach to partitioning and mapping makes the processor identifier explicit, which is exactly what is needed to determine statically whether the data is local or remote.

II. RELATED WORK

This section describes the high-performance computing world's existing automatic parallelization. While the world of high-performance computing can find a wonderful amount of work in automatic parallelization, there is little work on parallelizing compilers for Multi-DSP. Gupta et al. (1999) provides a good overview of existing parallelization techniques. The sub-theme of Chandra et al. (1997) is a cache-coherent multiprocessor with distributed shared memory. Although compilers for such machines must incorporate increasing techniques for data distribution and data locality (Carr et al., 1994; Tseng et al., 1995), they are not confronted with the problem of multiple address spaces that can be addressed globally. Compiler-Implemented Shared Memory (CISM) as described by Larus (1993) and Hiranandani et al. (1992) is a method for determining shared memory on computers that pass messages. These approaches, however, assume separate spaces for distributed addresses and require complex bookkeeping runtime. Paraguin (Ferner, 2003) is a compiler generating message-passing code, but this compiler is still in its infancy and requires directives from users to drive its parallelization. An early paper (Teich and Thiele, 1991) described how to parallelise DSP programs but did not provide details or experimental results. Similarly, an interesting overall parallelization framework is described in Kalavade et al. (1999), but no mechanism or details are provided as to how parallelization could occur. The impact of various parallelization techniques is considered in Lorts (2000), however, this has been user-driven and no automatic approach has been provided. A semi-automatic parallelization method is presented in Karkowski and Corporaal (1998) to allow design-space exploration of various multi-processor configurations based on MOVE architecture. However, there was no integrated data partitioning strategy and centralized data retention. In addition, communication was modelled in their simulator in the experiments, thus not addressing the issue of mapping parallelism combined with distributed address spaces.

III. PROPOSED WORK

In this section, we propose the partitioning approach and a new data address resolution technique in this section. Data arrays are partitioned along those array proportions which can be assessed in parallel and minimize communication. Determining those index proportions that can generally be evaluated in parallel provides a number of options and thus a simple technique is used to reduce data alignment-based communication. If two array references have each element corresponding to the same index space points in a certain proportion then they will be aligned. This means that[i][j] andb[i][k] are aligned with the first index but not with the second index. If two arrays are aligned with a particular index, then regardless of how those individual array elements are partitioned, any reference to that index will always be local. Alignment-based partitioning attempts to maximize the rows in a subscript matrix that are equal.
Let's define d(x;y) as follows:

$$\delta(x,y) = \begin{cases} 1 & x = y \wedge x \neq 0 \\ 0 & otherwise \end{cases}$$

This function determines whether two subscripts are non-zero and equal. The function H(i) defined as:

$$H(i) = \sum_{t} \delta(\mathcal{U}_i^1, \mathcal{U}_i^t)$$

Which measures how well Ut is used by a particular index I of an array is aligned with U1. The value of H is calculated for each index, the index with the highest value being the one accompanying the partition.

This technique is applied across all statements, and requirements for partitioning will generally conflict. Currently, only those statements in the deepest nested loops are considered as they dominate execution time and calculate H's value for different parallel indices I across all these statements. Imax H, the index with the highest value for H, determines the index along with the partition.

A partition matrix P is constructed:

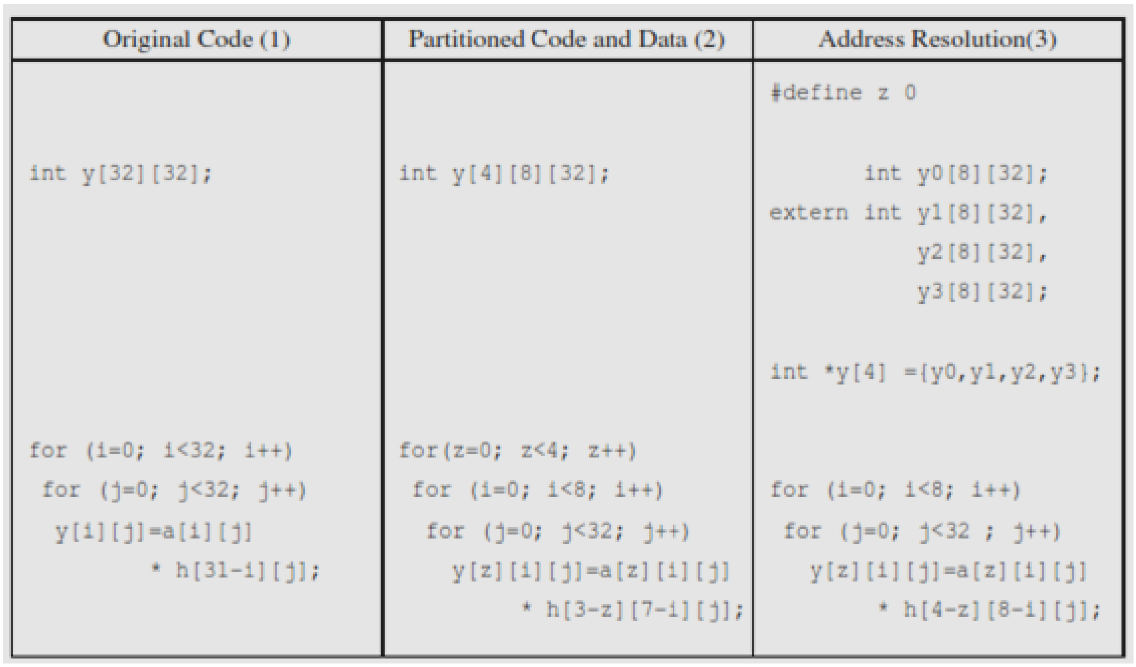
$$\mathcal{P}_i = \left\{ \begin{array}{ll} e_i^T & \text{if } i = i_{max,H} \\ 0 & \text{otherwise} \end{array} \right\}$$

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 8, Issue 7, July 2019

Where eTi is the ID matrix Id. ith row Also a sequential matrix S containing unpartitioned indices such that P + S= I d is built.



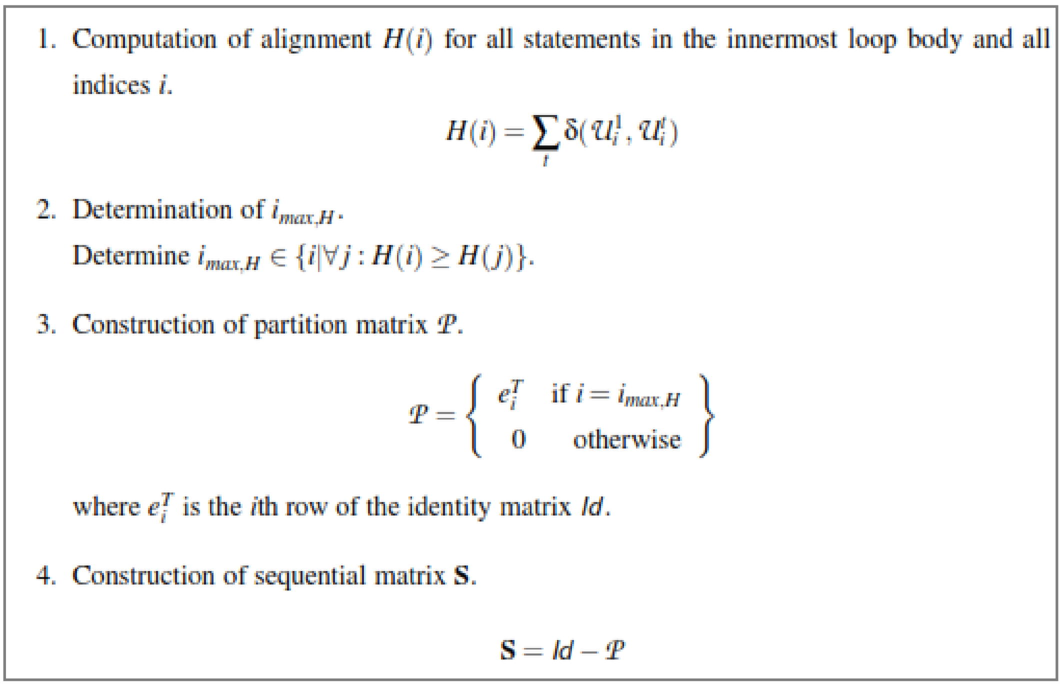
Original Code (1)	Partitioned Code and Data (2)	Address Resolution(3)

`int y[32][32];`
 	
`for (i=0; i<32; i++)`
` for (j=0; j<32; j++)`
`  y[i][j]=a[i][j]`
`         * h[31-i][j];`
	`int y[4][8][32];`
	
	`for(z=0; z<4; z++)`
	` for (i=0; i<8; i++)`
	`  for (j=0; j<32; j++)`
	`    y[z][i][j]=a[z][i][j]`
	`         * h[3-z][7-i][j];`
		`#define z 0`
		
		`       int y0[8][32];`
		`extern int y1[8][32],`
		`           y2[8][32],`
		`           y3[8][32];`
		
		`int *y[4] ={y0,y1,y2,y3};`
		
		`for (i=0; i<8; i++)`
		` for (j=0; j<32 ; j++)`
		`   y[z][i][j]=a[z][i][j]`
		`        * h[4-z][8-i][j];`

Figure 2. Partition and translation with communication of array h

In the example in figure 7.5, box(1), H(0) = 1; H(1) = 2; I max;H = 1 and therefore

$$\mathcal{P} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \text{ and } \mathbf{S} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$



1. Computation of alignment $H(i)$ for all statements in the innermost loop body and all indices i.

$$H(i) = \sum_{t} \delta(\mathcal{U}_i^1, \mathcal{U}_i^t)$$

2. Determination of $i_{max,H}$.
   Determine $i_{max,H} \in \{i \mid \forall j : H(i) \geq H(j)\}$.

3. Construction of partition matrix $\mathcal{P}$.

$$\mathcal{P} = \left\{ \begin{array}{ll} e_i^T & \text{if } i = i_{max,H} \\ 0 & \text{otherwise} \end{array} \right\}$$

   where e_i^T is the ith row of the identity matrix Id.

4. Construction of sequential matrix $\mathbf{S}$.

$$\mathbf{S} = Id - \mathcal{P}$$

Figure 3. Partitioning Algorithm

Once the array indices to partition along have been determined, strip-mining the indices J based on the partition matrix P and strip-mine matrix Sp produces the new domain J where Sp is defined as

$$S_p = \begin{bmatrix} (.)p \\ (.)\%p \end{bmatrix}$$

and p is the number of processors. Embedding of Sp produces a generalised stripmine matrix S. For further details see O'Boyle and Knijnenburg (2002). Let T be the mapping transformation where T = PS + S

Thus the partitioned indices are strip-mined and the sequential indices left unchanged. The new indices are given by

$$\boldsymbol{J'} = T\boldsymbol{J}$$

The new array bounds are then found:

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} AT^{-1}\boldsymbol{J'} \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} \boldsymbol{a}$$

and array accesses are updated accordingly

$$\mathcal{U}' = T\mathcal{U}$$

In general, without any further loop transformations, this will introduce mods and divs into the array accesses. However, by applying a suitable loop transformation, in this case T, this can be recovered.
Applying T to the enclosing loop iterators and updating the access matrices gives

$$\boldsymbol{I'} = T\boldsymbol{I}$$

where

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} BT^{-1}\boldsymbol{I'} \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} \boldsymbol{b}$$

And

$$\mathcal{U}'' = T\mathcal{U}T^{-1}$$

1. Construction of the *Transformation Matrix* T.

   (a) Construction of the *Strip-Mine Transformation Matrix*

   $$S_p = \begin{bmatrix} (.)/p \\ (.)\%p \end{bmatrix}$$

   and its *Pseudo-Inverse*

   $$S_p^\dagger = \begin{bmatrix} p & 1 \end{bmatrix}$$

   where p is the number of processors.

   (b) Construction of the *Generalised Strip-Mining Matrix* S and its *Pseudo-Inverse* $S^\dagger$

   $$S = \begin{bmatrix} Id_{k-1} & 0 & 0 \\ 0 & S_l & 0 \\ 0 & 0 & Id_{N-k} \end{bmatrix} \quad \text{and} \quad S^\dagger = \begin{bmatrix} Id_{k-1} & 0 & 0 \\ 0 & S_l^\dagger & 0 \\ 0 & 0 & Id_{N-k} \end{bmatrix}$$

   (c) Construction of the *Transformation Matrix* T.

   $$T = \mathcal{P}S + \mathbf{S}$$

2. Computation of new *Array Index Space*.

   (a) Computation of new *Array Indices* $\boldsymbol{J}'$.

$$\boldsymbol{J}' = T\boldsymbol{J}$$

   (b) Computation of new *Array Bounds* $A'\boldsymbol{J}' \leq \boldsymbol{a}'$.

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} AT^{-1}\boldsymbol{J}' \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} \boldsymbol{a}$$

3. Computation of new *Iteration Space* and *Loop Bounds*

   (a) Computation of new loop iterators $\boldsymbol{I}'$.

$$\boldsymbol{I}' = T\boldsymbol{I}$$

   (b) Computation of new loop bounds $B'\boldsymbol{I}' \leq \boldsymbol{b}'$.

$$\begin{bmatrix} T & O \\ O & T \end{bmatrix} BT^{-1}\boldsymbol{I}' \leq \begin{bmatrix} T & O \\ O & T \end{bmatrix} \boldsymbol{b}$$

4. Update of *Array References* $\mathcal{U}''\boldsymbol{I}' + \boldsymbol{u}$.

   Update all array references $\mathcal{U}'' = \mathcal{U}'T^{-1} = T\mathcal{U}T^{-1}$.

Figure 4. Mapping Algorithm

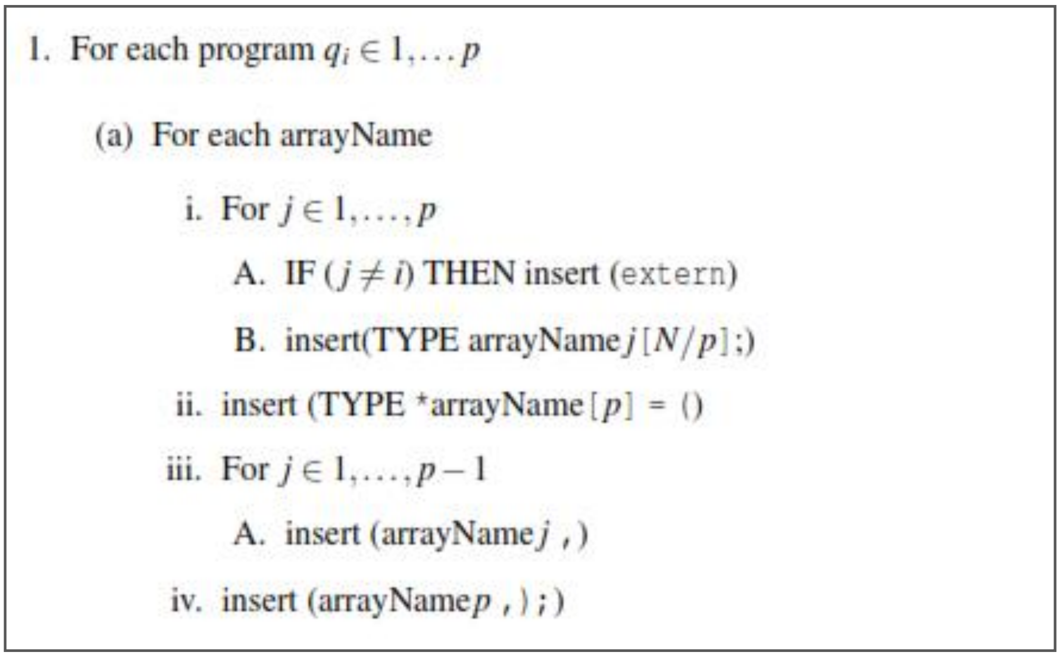
There are two problems with partitioning multiple address space architecture such as TigerSHARC: (a) separate local name spaces for variables, and (b) remote data addressing. The first issue can be solved by giving the partitioned arrays different names as follows. Once an array is partitioned and mapped across multiple processors, to distinguish it from other partitions on other processors, each local partition must be given a local name. Therefore a new name is introduced equal to the old one suffixed by the number of the processor identity. Thus, Z will be replaced by four local arrays Z0; Z1; Z2; Z3 in a four-processor system. The remote array declarations must be changed to external to make one of these arrays local and the other remote but still accessible. The linker will solve the addresses of such external arrays.

The second problem arises when translating a reference, e.g. x[i], into the new partitioned array form. The original reference always refers to array a, whereas the new reference must be able to refer to the x0 arrays that might be remote;::;x3. This problem is solved by introducing a small lookup table containing start addresses of the various partitions in the array. To determine the corresponding array, each access will refer to this table before the actual access is executed. The following section provides a complete description of the address resolution algorithm.

A pointer array of size p is introduced to minimize the impact on code generation, pointing to the start address of each of the p subarrays. Unlike the sub-arrays, this pointer array is replicated throughout the p processors and initialized at the start of the program with an array initialization statement. Figure 5 shows the complete algorithm where the function inserts the statements. Figure 2, box (3), shows the statements for one of the arrays inserted, y. Due to space, the statements for the remaining arrays are omitted. The only additional change is the declaration of type whenever the arrays are transferred to function. The declaration of type is changed from int[][] to* int[] and must be propagated inter-procedural. No further transformation or code modification is required once this has been applied.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

Vol. 8, Issue 7, July 2019



```
1. For each program $q_i \in 1,\dots p$
    (a) For each arrayName
        i. For $j \in 1,\dots,p$
            A. IF $(j \neq i)$ THEN insert (extern)
            B. insert(TYPE arrayName$j$[$N/p$];)
        ii. insert (TYPE *arrayName[$p$] = ()
        iii. For $j \in 1,\dots,p-1$
            A. insert (arrayName$j$ ,)
        iv. insert (arrayName$p$ ,);)
```

Figure 5. Address Resolution Algorithm

IV. CONCLUSION

In this paper we discussed a new parallelization approach for compilers that has been developed to map C programs to multiple address space multi-DSPs. For architectures with multiple visible address spaces, existing approaches to parallelization are not well suited. Single-address space such as parallel code can be generated for multiple address space architecture by using a novel data transformation and address resolution mechanism. The generated code is easy to read and easy to further optimize sequentially. A new parallel compiler approach has been developed to map C programs to multiple multi-DSP address spaces.

REFERENCES

[1]. Chandra, R., Chen, D.-K., Cox, R., Maydan, D., Nedeljkovic, N., and Anderson, J.(1997). Data distribution support on distributed shared memory multiprocessors. In Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI '97), pages 334–345, Las Vegas, NV, USA.
[2]. Carr, S., McKinley, K., and Tseng, C.-W. (1994). Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), pages 252–262, San Jose, CA, USA.
[3]. Ferner, C. (2003). Paraguin compiler for distributed systems. Available online: http://people.uncw.edu/cferner/Paraguin/.
[4]. Gupta, R, Pande, S, Psarris, K, & Sakar, V. (1999). Compilation techniques for parallel systems. Parallel Computing, 25(13–14), 1741–1783.
[5]. Hiranandani, S, Kennedy, K, & Tseng, C-W 1992. Compiling Fortran D for MIMD distributed-memory machine. Comm of the Acm,35(8), 66–80
[6]. Kalavade, A., Othmer, J., Ackland, B., and Singh, K. (1999). Software environment for a multiprocessor DSP. In Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC '99), New Orleans, LA, USA.
[7]. Karkowski, I. and Corporaal, H. (1998). Exploiting fine- and coarse-grain parallelism in embedded programs. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT '98), pages 60–67, Paris, France.
[8]. Larus, J. 1993 Compiling for shared-memory & message-passing comp. ACM Letters on Programming Language & Systems, 2(1–4), 165–180.
[9]. Tseng, C.-W., Anderson, J., Amarasinghe, S., and Lam, M. (1995). Unified compilation techniques for shared and distributed address space machines. In Proceedings of the International Conference on Supercomputing (ICS '95), pages 67–76, Barcelona, Spain.
[10]. Teich, J. and Thiele, L. (1991). Uniform design of parallel programs for DSP. In Proceedings of IEEE International Symposium Circuits and Systems (ISCAS '91),pages 344a–347a, Singapore.
[11]. Rijpkema, E., Deprettere, E., and Kienhuis, B. (1999). Compilation from Matlab to process networks. In Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES '99), Wyndham City Centre, Washington, DC, USA.
[12]. Lee, E. (1995). Dataflow process networks. Proceedings of the IEEE, 83(5), 773–801.