The ETCA Data-Flow Functional Computer for Real-Time Image Processing

Georges M. QUÉNOT*, Bertrand ZAVIDOVIQUE
Laboratoire Système de Perception
DGA/Etablissement Technique Central de l'Armement
94114 ARCUEIL CEDEX (FRANCE)

Abstract

This paper presents a data-flow computer, constituted of a large array of data-flow processors and programmed using a functional language, and its application to real-time image processing.

The approach integrates tightly and efficiently the data-flow architecture principle and the functional programming concept. It leads to a very simple and regular hardware. It also provides a very efficient and user-friendly software interface for application development. It remains applicable in systems that includes processors of different granularity.

An experimental system, including 1024 low-level custom data-flow processors (6 × 25 MBytes/s, 50 million operations per second) and 3 'T800' transputers, was built and several image processing algorithms were run in real time at digital video speed.

1: Introduction

This paper describes the general concept of a data-flow computer programmed in a functional style and a practical application of this concept for real-time image processing.

Real-time image processing requires a very high computing power (in the range of billions of operations per second). It also requires heterogeneous architectures since a single type of processor cannot handle efficiently all the levels (low, intermediate or high) of image understanding algorithms [1]. Several prototypes of vision machines exist, but they are very complex and very hard to program.

The goal was to develop a very high performance computer architecture, able to integrate different types of processors and easily programmable. The approach has been to unify the hardware concept of data-flow architectures [2][3] and the software concept of functional programming [4]. Fig. 1 summarizes this approach on a very simple example. By doing so, it has been possible to design a massively parallel and very regular computer mixing fine grain and coarse grain processors. It has also been possible to build an homogeneous, natural and easy-to-use software environment for this machine.

2: Data-Flow Computer Architecture

According to a classical taxonomy for parallel computer architectures [5], the presented machine can be classified as a data-flow computer and, more precisely, as a wavefront array-processor [6]. It can also be considered as an actual implementation of a MISD computer since all the operations involved in an algorithm are performed in parallel (in a pipe-line mode) on a single digital video data-stream. It can also be noticed that the use of the data-flow concept is a "pure" one since there is no address flow in this machine.

Data-flow architectures have been proposed as alternatives to Von Neumann's architectures and as a way to remove associated bottlenecks [2][3]. However, classical data-flow architectures also appear to have their own bottlenecks, especially in the cross-bar network that they must include and with the load distribution problem [7][8]. At the price of some loss in generality, those last bottlenecks were removed. The presented approach is well suited for (and, in general, only for) an *iterative processing* of *a large number of elements, all of the same type,* and when *a fixed number of operations have to be performed for each element.* This is the case for low-level image processing. For example: a 5×5 convolution on an image requires 25 multiplications and 24 additions for each pixel.

A "data-flow" is a *structured data set moving serially along a physical link* ("data-stream" would be more accurate in this context). A "Functional parallelism", where all the operations corresponding to one iteration are executed in parallel for only one element at every time step (MISD), is used instead of a "Data parallelism", where only one operation is performed for all the elements at every time step (SIMD, as this is the case for the Image Understanding Architecture [1] or the CM2 Connection Machine). There is a one-to-one correspondence between physical processors and operations in the algorithm instead of a one-to-one correspondence between physical processors and data elements. For the 5×5 convolution 25 processors are used as multipliers and 24 processors are used as adders. A few other processors are also necessary for providing horizontal and vertical shifts and for data-flow routing.

Unlike in other classical data-flow architectures that also use functional parallelism, program execution is kept very structured by setting the constraint that a given processor will always execute the same operation (MISD instead of MIMD).

The core of the data-flow functional computer is a regular network of data-flow processors. Data-flow processors include a cross-bar circuit and are used simultaneously as data-flow operators and as data-flow routing

* Now at LIMSI-CNRS, Orsay, France.

ALGORITHM:

Y(n) = a.X(n+1) + b.X(n) + c.X(n-1)

INVOLVED FUNCTIONAL OPERATORS:

D : (X(1) X(2) . . . X(n)) = (0 X(1) . . . X(n-1))
id : (X(1) X(2) . . . X(n)) = (X(1) X(2) . . . X(n))
A : (X(1) X(2) . . . X(n)) = (X(2) X(3) . . X(n) 0)

xa : (X(1) X(2) . . . X(n)) = (a.X(1) a.X(2) . . . a.X(n))

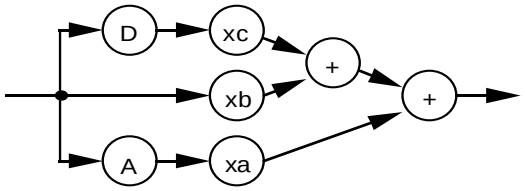
+ : ((X(1) X(2) . . . X(n)) (Y(1) Y(2) . . . Y(n))) =
(X(1)+Y(1) X(2)+Y(2) . . . X(n)+Y(n))

FUNCTIONAL EXPRESSION:

DEF F = +o[xa o A, +o[xb o id, xc o D]]

(Y(1) Y(2) . . . Y(n)) = F : (X(1) X(2) . . . X(n))

DATA-FLOW GRAPH:



MAPPING ONTO THE DATA-FLOW COMPUTER:

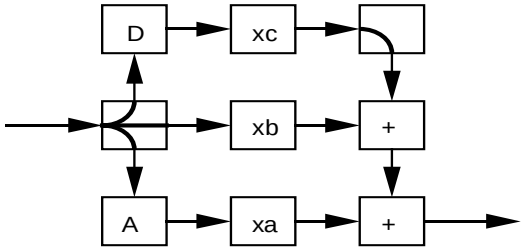


Figure 1: Functional programming and data-flow implementation: A simple example

elements. Several types of processors may be used in the network provided that each type can communicate with one another (the practical implementation incorporates two types of processors: a fine grain custom processor and a coarse grain T800 processor).

A single operator is assigned to each processor. The program of a processor is made of two distinct and independent parts. The first one defines the function of the operator. The second one defines the routing of data-flows between the physical ports and the logical input-outputs of the operator. This applies to all types of processors.

The execution of programs on the data-flow functional computer is fully data-driven. There is no global control. Each operator is activated by a completely local firing rule, and operations are performed as soon as all the required data are available and if, simultaneously, destination processors for output flows are ready to receive new data.

An application algorithm is defined as a complete data-flow graph whose nodes correspond to basic operators. This data-flow graph is then mapped onto the physical network. The algorithm is executed through a gigantic pipe-line. Its complexity is therefore limited by the size of the available array but when it fits in, it is always executed at the same speed (real-time digital video in this application) whatever its complexity.

This approach appears to imply that all atomic operations should have the same complexity and this could seem to lead to inefficient implementations. This is one of the reason for which the architecture must include processors of different granularity. This effect is also attenuated since several simple operators can be associated to a single processor and complex operators can be distributed as macro-operators among a small group of processors.

Using a powerful processor only for routing purposes may also appear inefficient but cross-bar chips are always necessary in data-flow architectures and including a CPU within them does not cost much at the system level. The benefits associated with the system simplicity and regularity compensate for the loss due to poor CPU use yields. Moreover, all actually working CPUs run at full speed (given the restriction that an expression like "if A then B else C" is computed as "A.B + (1−A).C", in such a case B and C are always both computed).

3: Functional Programming

Classical programming languages (Von Neumann style ones like C or FORTRAN) appeared very poorly suited for the data-flow machine. Alternative programming languages like functional programming [4] or LISP have been proposed and appeared to have very good properties for data-flow graph descriptions [9]. In order to efficiently couple the functional programming concept to the data-flow approach, some limitations were also applied to it.

The first one is that all the basic primitive functions of the language must correspond to an operator implementable in real-time on at least one type of processor of the data-flow computer.

The second one is that only two functional forms are allowed for the construction of new functions from existing ones:

The composition form defined by:

$$(f \circ g) : x = f : (g : x)$$

The construction form defined by:

$$[f_1, ..., f_n] : x = (f_1 : x, ..., f_n : x)$$

This is not a big limitation since most of other functional forms can be placed at the basic function level. For example: f, $/f$ (Insert form) and $\alpha.f$ (ApplyToAll form) can be distinct basic functions.

The third one is rather a generalization. Normally, a functional program takes only one input object and provides only one output object. The extension is to

admit that a program (and actually any function) may take several inputs and provide several outputs (this practical implementation detail does not have any consequence from the theoretical point of view).

By doing so, the transformation between a functional expression and a data-flow graph is straightforward. All operators have simultaneously a software aspect (primitive function) and a hardware aspect (physical operator). The composition functional form corresponds to a serial placement of operators and the construction functional form corresponds to a parallel placement of operators (Fig. 1).

A library has to be defined for each type of processor that appears in the network. Normal users will not generally need to develop their own basic operators since the machine will be delivered with a software package that includes a large library of currently used operators and macro-functions (exactly as normal users do not have to write machine language code on classical computers). Users will only need to express their algorithms with the functional programming language.

This approach has the advantage of introducing a software hierarchy that greatly reduces the programming complexity. It also provides an homogeneous software interface for an heterogeneous architecture (although the architecture is made of a regular array of processors, it is heterogeneous since it includes several types of processors).

Programming the data-flow functional computer is then easy and efficient. Users may describe their algorithms in a text form or in a graphic form. They may place and route their data-flow graphs themselves using a graphic editor or let the system do it (of course, a manual placement is more efficient).

This use of Functional Programming is also a "pure" one since all algorithms can be described by functional expressions without the concept of variable (Fig. 1). This is the software counterpart of the elimination of the address concept at the hardware level. This common elimination of the variable and address concept reflects the very strong relationship that exists between "pure" functional programming and "pure" data-flow architectures.

4: Image Processing Application

The target application was real-time image processing. Therefore, the data-flow computer incorporates two types of processors. The first one is dedicated to low-level image processing (one operation per pixel). The second one is dedicated to high-level image processing (a fixed or bounded number of operations per image). The data-flow computer also incorporates digital video input/output subsystems and a global controller coupled with a host workstation (Fig. 2).

For the execution of low-level functions, a custom data-flow processor (DFP) was developed. Two coupled DFPs have been included in a single VLSI chip which is fully described in [10]. Each DFP contains 6 input-output ports, a complete crossbar, 3 input FIFO stacks and 3 output FIFO stacks, a datapath, a 256

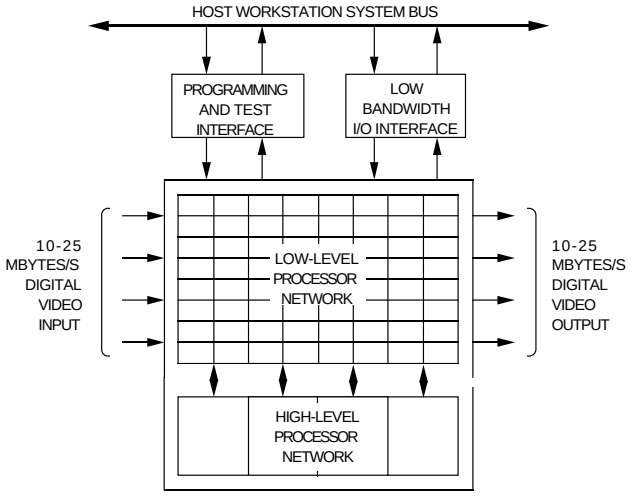


Figure 2: The data-flow computer architecture

8-bit word data RAM and a 64 32-bit word program RAM. DFPs are able to process on the fly 25 MBytes/s data-flows and to perform up to 50 million 8/16-bit operations per second.

A wide variety of data-flow operators can be implemented on the DFPs. Among them are additions, subtractions, multiplications, line or pixel shifts, derivatives, summations, scalar products and histograms. More complex operators (convolution for example) can be defined as macro-functions using combinations of basic operators mapped onto groups of processors.

For the execution of high-level functions, the T800 transputer was chosen because it is a general purpose processor which already incorporates communication links. It is therefore very well suited for the implementation of complex functions into data-flow operators. Transputer modules including 1 MByte of memory are the basic elements of the high-level processing network.

Any kind of algorithm may be implemented on a high-level processing element as a C program. There are only two constraints: the first is to follow the data-flow principles (output data-flows are only functions of input data-flows and control is done locally, i.e. execution is independent of what may occur in other processors) and the second one is to satisfy real-time.

The cross-bar that should be included in each transputer node is emulated by software. For this, a common specific header is included in every operator source file.

The following table summarizes the characteristics of both processors and shows the duality between them. Low-level and high-level processors networks communicate using bidirectional parallel/serial link adaptors.

DFP 8-bit	T800 32-bit
Low-level Simple operators	High level Unlimited complexity
High bandwidth 6×25 MBytes/s 50 Mops	Low bandwidth 4×2 MBytes/s 3 Mflops

The prototype data-flow functional computer in-

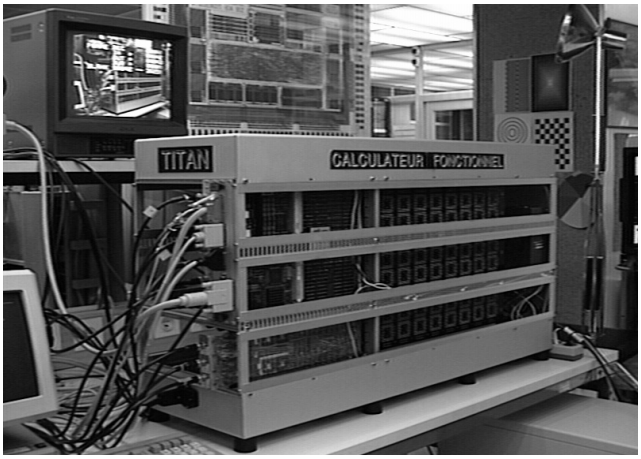


Figure 3: The ETCA data-flow functional computer

cludes 1024 DFPs (with 512 bi-processor chips) in a $8 \times 8 \times 16$ tri-dimensional network and 3 T800 transputer (Fig. 3). It will be extended soon to 36 T800 based transputer modules in four 3×3 bi-dimensional networks.

A complete software environment has been developed. It includes: a Data-Flow Functional Programming (DFFP) language compiler which translates a DFFP source file into a Data-Flow Graph (DFG) of operators, a mapper that places and routes a DFG onto the physical network and a graphic editor for manual modification or optimization of the physical DFG. A preliminary version of a library of operators has been developed for both types of processors.

5: Performance evaluation

By nature, this computer operates at a fixed speed (in this case at digital video speed, up to a 25 Mhz pixel frequency). Therefore, its performance cannot be evaluated by the time it takes to perform a given task. The performance measure should rather be the number of processors (or the fraction of the machine) needed to perform a given task.

More precisely, the performance index is the number of operations per pixel performed at digital video speed. Each DFP can perform 1 or 2 operations per pixel. Then, the maximum theoretical power of the computer is 2000 operations per pixel. Depending upon the regularity of the algorithms and the degree of placement optimization, the practically available power ranges from 200 to 800 operations per pixel. A complete benchmark has not been performed yet since the operator library is still under optimization.

This computer is not limited by I/O problems. Using the implemented I/O channels it can input, process on the fly and output simultaneously two black and white and two color digital video streams.

The most complex algorithm that has been implemented until now involves about 120 operations per pixel. It performs the extraction of a set of parallel lines, based on the computation of the histogram of gradient direction and of a partial Radon transform. It was executed in real time on 25 Hz sequences of 800×572 images (16 MHz pixel frequency).

This represents about 2 billion 8/16-bit operations per second and was achieved with 1/4th of the complete configuration of the computer. In order to run the same algorithm in real-time with a CM2 connection machine, a 32K configuration would be required.

6: Conclusion

The concept of a data-flow functional computer (DFFC) developed at ETCA and its application to real-time image processing have been described. The approach integrates efficiently the data-flow architecture principle and the functional programming concept together. It leads to a very simple and regular hardware. It also provides an efficient and user-friendly software interface for application development. It can be maintained even with a system that includes processors of different granularity.

A prototype data-flow functional computer with 1024 low-level DFPs and 3 T800 transputer modules was built. Several significant image processing algorithms were successfully implemented in real time.

The implementation of a data-flow functional computer presented in this paper is mainly dedicated to image processing. However, its concept is much more general. For example, a 64-bit floating point data-flow processor could similarly be developed and used for the design of a data-flow functional supercomputer dedicated to the resolution of partial derivative equations by finite difference methods.

References

[1] C.C. Weems et al.,"The Image Understanding Architecture," *Int'l J. Computer Vision,* Vol 2, No 1. Jan 1989, pp 251-282.

[2] J.B. Dennis, "Data Flow Supercomputers," *Computer,* Vol 13, Nov. 1980, pp. 48-56.

[3] K.P. Arvind, D.E. Culler, "Dataflow Architectures," Annual Reviews In Computer Science, Vol. 1, Palo Alto CA, Annual Reviews Inc 1986, pp. 225,253.

[4] J. Backus, "Can programming be liberated from Von Neumann Style? A functional style and its algebra of programs." *Comm. of ACM,* Vol. 21, No 8, August 1978.

[5] R. Duncan, "A Survey of Parallel Computer Architectures," *Computer,* Vol. 23, No.2, Feb. 1990, pp 5-16.

[6] S.Y. Kung et al., "Wavefront Array Processors – Concept to Implementation," *Computer,* Vol. 20, No.7, July 1987, pp 18-33.

[7] J.R. Gurd, C. Kircham and I. Watson, "The Manchester Prototype Dataflow computer," *Comm. of ACM,* Vol. 28, No 1, January 1985.

[8] K.P. Arvind and R.S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture" *IEEE Transactions on computers* Vol. 39, No. 3. March 1990.

[9] D. Genin et al., "DSP Specification using the Silage Language," *IEEE ICASSP'90,* Vol. 2, pp 1057-1060, April 1990, Albuquerque, NM.

[10] G.M. Quénot, B. Zavidovique, "A Data-Flow Processor for Real-Time Low-Level Image Processing," *IEEE Custom Integrated Circuits Conference,* May 1991, San-Diego, CA.