FPGA implementation of the k-means clustering algorithm for hyperspectral images

Dominique LAVENIER

Los Alamos National Laboratory
NIS-3, Space Data Systems
Los Alamos - NM 87545 - USA

July 2000

LA-UR # 00-3079

Abstract

This document describes a parallel architecture for computing the k-means clustering algorithm based on a pixel data stream. The implementation targets FPGA accelerator boards connected to a host processor through a standard I/O interface. Speed-up for hyperspectral image processing is estimated on a systolic processor array based implementation.

1 Introduction

The basic principle of the image clustering process is to take an original image and to represent the same image using only a small number of pixel values. The k-means clustering algorithm [1], [2] performs this task by attempting to minimize a squared error cost function over a set of NB_CLASS cluster centers. The k-means algorithm works as follows:

- Assign pixels to NB_CLASS classes and Compute centers *initialization*
- Loop (**N**) *k-means iteration*
  - For each pixel:
    - **C** = class of the pixel
    - Determine the class number **K** whose centers have the smallest distance from the pixel
    - If (**C != K**)
      - Move pixel to class **K**
      - Recompute the centers of the classes **K** and **C**

The number of iterations (**N**) can be either fixed in advance or undetermined. In that case, the process stops when no more pixels, or less than a threshold value, move.

Actually, the k-means algorithm includes several variants ranging from a continuously class center updating (as mentioned above) to a general updating (once after a complete loop iteration). But more often the class centers updated, faster the convergence.

2 Algorithm profiling

As we want to implement the k-means algorithm (or part of this algorithm) into hardware, it is first worth to study its behavior to determine where are the costly sections. The following code is a C transcription of the k-means iteration:

```
1  while (pixel_move !=0) {
2  pixel_move = 0;
3    for (i=0; i<NB_PIXELS; i=i+B) {
4      for (b=0; b<B; b++) {
5        min = MAX_INT;
6        /* compute a distance between the pixel and all the classes  */
7        for (k=0; k<NB_CLASS; k++) {
8          if (N_CENTER[k]!=0) { cntr_dist += 1;
9            dist = 0;
10           for (d=0; d<NB_BANDS; d++)
11             dist = dist + ABS (PIXEL[i+b][d] - CENTER[k][d]);
12           /* get the minimum distance and the associated class # */
13           if (x<min) { min = dist; idx[b] = k; }
14         }
15       }
16     }
17     for (k=0; k<NB_CLASS; k++) change[k] = false;
18     for (b=0; b<B; b++) {
19       if (CLASS[i+b]!=idx[b]) { cntr_acc += 2;
20         pixel_move ++;
21         k = CLASS[i+b]; N_CENTER[k]--; change[k] = true;
22         for (d=0; d<NB_BANDS; d++)
23           ACC[k][d] = ACC[k][d] - PIXEL[i+b][d];
24         k = idx[b]; CLASS[i+b] = k; N_CENTER[k]++; change[k] = true;
25         for (d=0; d<NB_BANDS; d++)
26           ACC[k][d] = ACC[k][d] + PIXEL[i+b][d];
27         CLASS[i+b] = idx[b];
28       }
29     }
30     for (k=0; k<NB_CLASS; k++)
31      /* recompute centers if necessary */
32       if (N_CENTER[k]!=0 && change[k]==true) { cntr_center += 1;
33         for (d=0; d<NB_BANDS; d++)
34           CENTER[k][d] = ACC[k][d]/N_CENTER[k];
35       }
36    }
37  }
```

A loop iteration scans all the pixels. For each pixel we check if it belongs to its class. If not, the pixel is moved to another class and the two centers corresponding to both the new and the old classes are updated. The number of pixels into a class is memorized as well as the sum accumulation necessary for recomputing the class centers. Here, the class centers are periodically updated every block of B pixels.

One may also note that the distance function has been simplified to use the Manhattan instead of the Euclidean metric [1]: this uses the absolute value of a difference instead of the squared difference. Today, this is more suitable for hardware implementation, but tomorrow FPGA component will include specific hardware in such a way that multiplication will be a cheap hardware operation [8].

The computation can roughly be split into three parts: the distance calculation between a pixel and a class center, the accumulator update and the center update. The three counters `cntr_dist`, `cntr_acc`, and `cntr_center` (respectively inside the if statement at lines 8, 19 and 32) spy these activities: `cntr_dist` is incremented by one each time a new distance is calculated, `cntr_acc` is incremented by two each time the accumulators are changed and `cntr_center` is incremented by one each time a new center is updated.

We run this algorithm on a 224-band hyperspectral image of 240x256 pixels using different parameters such as the number of classes and the updating frequency of the class centers. The results are summarized in the following tables:

#class	4	8	16	32	64
dist	2 703 360	6 389 760	36 372 480	81 511 680	136 417 920
acc	111 410	134 808	296 016	198 428	245 436
center	5 235	7 708	34 041	37 422	64 335

B = 240

#class	4	8	16	32	64
dist	2 703 360	6 389 760	52 101 120	78 658 560	156 698 880
acc	111 342	134 598	213 624	213 138	247 122
center	13 998	18 635	63 277	63 314	98 739

B = 60

#class	4	8	16	32	64
dist	2 703 360	6 389 760	26 542 080	106 997 760	149 735 040
acc	111 338	134 578	182 538	242 434	270 690
center	31 803	39 579	70 746	117 470	148 804

B = 12

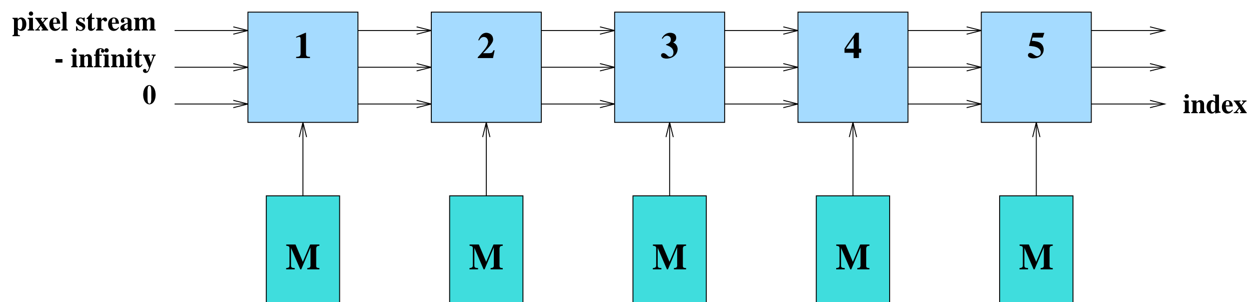
B represents the update frequency of the class centers. In other words, the class centers are updated after processing B pixels. The k-means process ends when there are no more pixels to move.

From these results, it is clear that the most consuming code section is the distance computation between the pixels and the class centers, even if the class center is updated very often. The accumulator and the class center updating represent only a small percentage of the total computation time, especially for a partition into a large number of classes. For example, in the case of a 32 class partitioning, the distance computation represents more than 99.6% of the computation time.

The architecture we describe in the next section focus only on parallelizing the most consuming part, that is the distance computation between the pixels and the class centers.

3 Parallel Architecture

The basic idea is to flow a pixel stream (from left to right) through a linear array of processors. The number of processors is equal to the number of classes. A processor **k** computes a distance between the class **k** and the current flowing pixel. The result is taken at the rightmost end of the array and corresponds to the index class for which a minimum distance has been found.



Each processor has a small memory storing the class center (a vector of NB_BAND values), and performs the following computation:

```
while (1) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    read (pixel);
    dist = dist + ABS(pixel - class_center[d]);
    write (pixel);
  }
  read (min, index);
  if (dist < min) {
    min = dist; index = #proc;
  }
  write (min, index);
}
```

This parallel structure does not compute any class centers, it allows only to determine what is the class number of a pixel. This information is available at the rightmost end of the array each time a pixel (or more precisely, its last vector element) comes out of the array. Consequently, we

suppose this array integrated in a digital system composed of a host processor capable of handling such operations. In other words, the host has the charge of flushing the pixel stream to the array and getting the results indicating the class number of each pixel. It is thus its responsibility to determine the moves of the pixels and to recalculate the class center accordingly.

At this point, we still haven't addressed the problem of updating the class centers: we suppose the processor memories loaded with the class centers before sending a block of **B** pixels through the array. The updated class center mechanism we have added benefits from the identical data structure between the pixels and the class centers: they are both a vector of NB_BAND data. The idea is to have a *heterogeneous* data stream composed of pixels and class centers. When a processor read a pixel it performs a distance computation (as described above), and when it read a class center it updates its memory. The program of a processor is modified as follows:

```
while (1) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    read (flag, data, min, index, class);
    if (flag  == pixel) {
      dist = dist + ABS(data - class_center[d]);
      if (dist < min) {
        dist = min; index = #proc#;
      }
    else if (index == class) class_center[d] = data;
    write (flag, data, dist, index, class);
  }
}
```

The variable `data` represents either a pixel or a class center depending of the value of the variable `flag`. If it is a pixel value, a distance is computed. If it is a class center value, then the `class_center` array is updated only if the `class` variable is equal to the processor index. The comparison between the `dist` and the `min` variables is now moved inside the **for** loop for "regularity" purpose, especially for inter processor communication: each iteration of the loop requires reading and writing exactly four data, independently of the nature of the stream elements (pixels or class centers).

In that scheme, a typical scenario is that first the host initializes the `class_center` array (processor memories) by sending NB_CLASS center class vectors. Then it sends a few pixels (a sub-stream of **B** pixels), read the indexes and determines if some pixels have moved. If so, it recalculates the new class centers and updates the processor memory by just sending the class centers which have changed. The size of the pixel block (**B**) is independent of the implementation and may vary from iteration to iteration.

4 Speed-up Evaluation

This section tries to evaluate the gain of connecting the linear structure described in the previous section. We suppose a reconfigurable platform connected to a host computer through a standard I/O interface (such as PCI). The main loop performed by the host computer becomes:

```
01   while (pixel_move !=0) {
02     pixel_move = 0;
03     for (i=0; i<NB_PIXELS; i=i+B) {
04       write_block_pixel (PIXEL[i],B);
05       for (k=0; k<NB_CLASS; k++) change[k] = false;
06       for (b=0; b<B; b++) {
07         read (idx);
08         if (CLASS[i+b]!=idx) {
09           pixel_move ++;
10           k = CLASS[i+b]; N_CENTER[k]--; change[k] = true;
11           for (d=0; d<NB_BANDS; d++)
12             ACC[k][d] = ACC[k][d] - PIXEL[i+b][d];
13           k = idx; CLASS[i+b] = k; N_CENTER[k]++; change[k] = true;
14           for (d=0; d<NB_BANDS; d++)
15             ACC[k][d] = ACC[k][d] + PIXEL[i+b][d];
16         }
17       }
18       for (k=0; k<NB_CLASS; k++) {
19         if (N_CENTER[k]!=0 && change[k]==true) {
20           for (d=0; d<NB_BANDS; d++)
21             CENTER[k][d] = ACC[k][d]/N_CENTER[k];
22           write_center (CENTER[k]);
23         }
24       }
25     }
26   }
```

The loop computing the distance is replaced by a procedure sending the pixels to the array (procedure write_block_pixel, line 04). Also, the processor memories are updated by sending new class centers each time they are modified (procedure write_center, line 22).

In section 2 we have shown that most of the time is spent in computing the distance between the pixels and the class centers. Assuming that the linear array can sustain the data rate provided by the host, the speed-up is mainly determined by the ratio $T1/T2$ where $T1$ is the computation time on a sequential processor and $T2$ is the computation time using the parallel array for computing the distances. More precisely:

$$T1 = Tdseq + Ta + Tc$$

$$T2 = Tdpar + Ta + Tc + Tc_{update}$$

$$Tdseq = cntr_dist \times NB_BANDS \times NB_CLASS \times (1/MEDS)$$

$$Tdpar = cntr_dist \times NB_BANDS \times (1/TR)$$

$$Tc_{update} = cntr_center \times NB_BANDS \times 1/TR$$

- $Tdseq$ is the time for computing the distance sequentially.
- $Tdpar$ is the time for computing the distance on the parallel array.
- Tc_{update} is the time for updating the class center on the array.

- Ta is time for updating the accumulators.
- Tc is the time for recomputing the class centers.
- $Tac = Ta + Tc$.
- $MEDS$ is a unit standing for Millions of Elementary Distances per Second.
- TR is the I/O transfer rate between the host and the array expressed in Mbytes per second.

Let $\alpha = Tac/Tdseq$ and $Sd = Tdseq/Tdpar$. α represents the ratio between the accumulator plus the center class update time and the distance computation time. Sd represents the speed-up of the distance computation. The overall speed-up is thus equal to:

$$S = Sd/(1 + \alpha \times Sd)$$

The details of calculation can be found in annex 1. Tests on a 450 MHz PC give a $MEDS$ of 9.7 (9.7 millions of elementary distances are computed every second).

5 Systolic Architecture

Among all the possibilities for implementing the array, we describe a systolic processor array. Figure 1 details the architecture of one processor. All the data and control signals are propagated synchronously. There are two control signals:

- **WrMem:** when active, the memory is loaded with a new value if the input IdxIn is equal to #proc.
- **LdAcc:** when active, reset the accumulator to the value output by the $|A - B|$ module.

The other input/output represent:

- **Class:** class number.
- **Addr:** memory address.
- **Pixel/Center:** depending of the value of WrMem the data flowing through this channel is a pixel (WrMem=0) or a class center (WrMem=1).
- **Idx:** index (valid only when WrMem=0).
- **Min:** minimum distance computed (valid only when WrMem=0).

For efficiency the systolic array cannot be directly connected to the host processor. A front process must be added for generating the control signals and the initialization values. This process has to perform the following task:

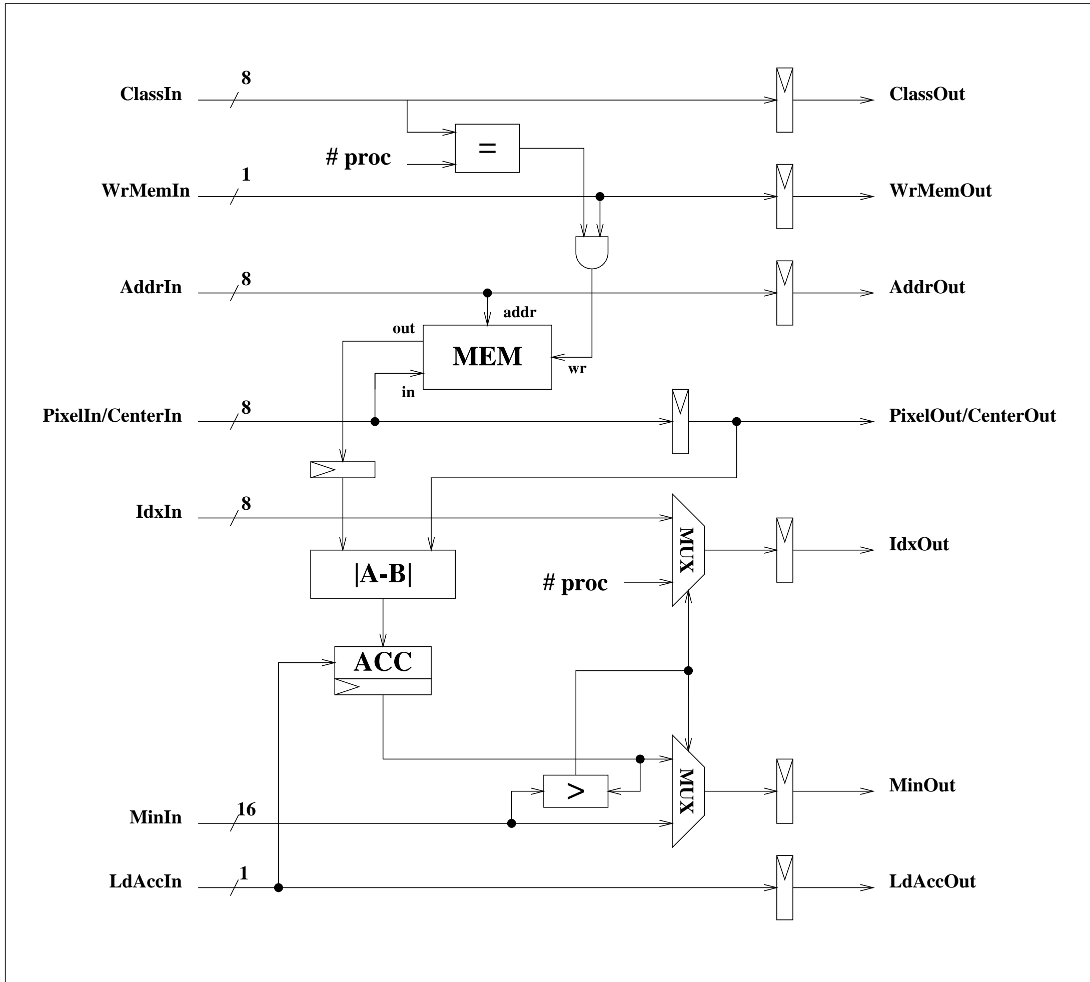


Figure 1: architecture of the systolic processor

```
while (1) {
  ReadVector (flag,vector);
  if (flag<0) {
    send (0,1,0,0,vector[0],0,MAX_INT);
    for (d=1; d<NB_BANDS; d++) send (0,0,0,d,vector[d],0,MAX_INT);
  }
  else
    for (d=0; d<NB_BANDS; d++) send (1,0,flag,d,vector[d],0,MAX_INT);
}
```

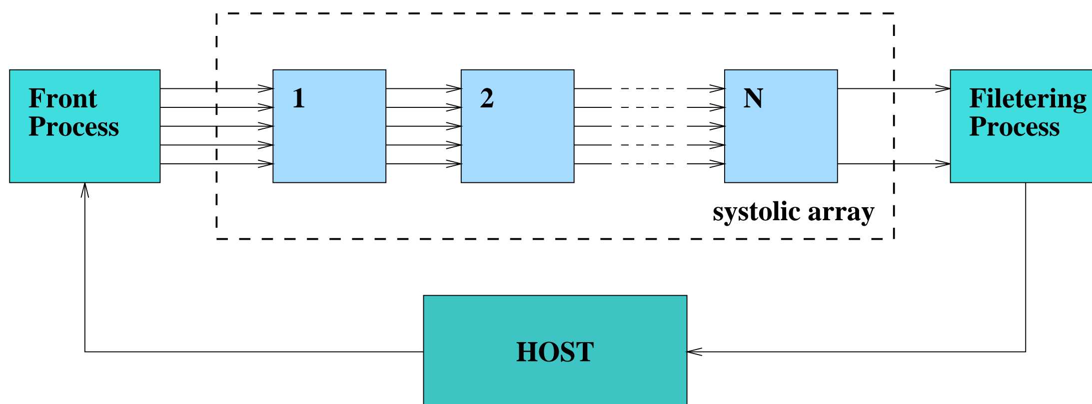
The infinite loop read a vector of NB_BANDS data which are either pixels or class centers. The vector is flagged with the `flag` variable: a null or positive value indicates a class center update. The procedure `send` as the following arguments:

```
send (WrMem, LdAcc, Class, Addr, Pixel/Center, Idx, Min)
```

On the same way, the rightmost systolic processor needs to be interfaced to the host processor by a process filtering the results:

```
d=0;
while (1) {
  read (WrMem, Idx);
  if ((WrMem==0)&&(d==NB_BANDS-1)) send2host (Idx);
  d = (d+1)%NB_BANDS;
}
```

The complete architecture is thus composed of a systolic array of NB_BANDS processors interconnected to the host as follows:



6 FPGA Implementation

The VHDL code source of a processor can be found in annex 2. The body includes 5 components (mem224x8, AinfB16, Equ8, Sub16, Acc16) designed with the Xilinx LogiBLOX tool. The synthesis process with Synplify and the place-and-route process with the Xilinx Alliance Series tools fit a processor (including its memory) into 120 CLBs for the XC4000 family [6]. The resulting clock frequency is equal to 20 MHz when a FPGA component with a few processors representing 80% of the available resources.

From this results we can estimate the speed-up for different FPGA boards using the speed-up expression given in section 4. In the following, we take a $MEDS$ value (Millions of Elementary Distances per Second) of 10. This corresponds roughly to the power computation capability of a 500 MHz processor. We also consider a pixel element encoded on 8 bits.

The α coefficient ($\alpha = Tac/Tdseq$) which estimates the ratio between the time spent for updating both the accumulators and the class centers, and the time for computing the distances is deduced from the tables given in section 2. We assume that the cost of the operation perform on the most inner loops (line 11, 23, 26, 34) are equivalent. Actually, this cost depends greatly of the accesses of the variables `PIXEL`, `CENTER` and `ACC`. The integer operations represents a small part of the computation time of the loops. Hence, α can be estimated from the counters values as the ratio of:

$$\alpha = \frac{cntr_dist}{cntr_acc + cntr_center}$$

Taking the worst case (B=12) for a number of classes ranging from 4 to 64 gives:

# of class	4	8	16	32	64
α	0.05	0.03	0.01	0.003	0.002

As the transfer rate between the host and the FPGA board is an important element, we take into consideration two situations:

- DMA transfer: In this case, the data transfer rate is not a limited factor. The speed-up is dictated by the clock frequency of the processor array. With a frequency of X MHz, the distance computation speed-up (Sd) is simplified to $0.1 \times X \times$ NB_CLASS.
- Memory Map transfer: This transfer mode is generally very slow and the speed-up is dictated by the host/board transfer rate. With a MEDS of 10, the distance computation speed-up (Sd) is equal to NB_CLASS $\times TR/10$.

Wildforce board

The Wildforce board [4] is composed of 5 Xilinx XC4036EX (36 x 36 CLBs) processing elements. An implementation can fit 8 systolic processors into each processing element PE1 to PE4 the front and filtering processes into PE0. This leads to an array of 32 processors. In the Memory Map transfer mode, the communication between the host and the board is very slow. A data bandwidth of 2-3 Mbytes/sec seems to be a maximum.

#class	4	8	16	32
MemMap Speed-up	0.7	1.5	3.1	6.3
DMA speed-up	5.7	11	24	53

SLAAC-1 board

The SLAAC-1 board [10] has twice the resources of the Wildforce board: up to 64 processors can then be implemented. Experiments carried out by the SLAAC-1 board designers give a transfer rate of 5 Mbytes/sec in the Memory Map mode. Currently, no DMA engine is available.

#class	4	8	16	32	64
MemMap Speed-up	1.8	3.6	7.4	15	30
DMA speed-up	5.7	11	24	53	101

Wildcard board

The Wildcard board houses a single Virtex 300 component [7]. This represents an array of 32 x 48 CLBs. Knowing that a Virtex CLB is approximatively equivalent to two XC4000 CLBs, a processor can be fitted into 60 Virtex CLBs. An array of approximately 20 processors (75% of CLBs used) can fit into this board running at a double frequency (40 MHz) compared to the Xilinx 4K family. The measured Memory Map transfer rate is 2.6 Mbytes/sec and a DMA write as a bandwidth of 40 Mbytes/sec.

#class	4	8	16
MemMap Speed-up	1	2	4
DMA speed-up	9	16	39

Wildstar board

The Wildstar board [5] interconnects three Virtex 1000 components (64 x 96 CLBs). Each component can fit easily 80 processors (75% of CLBs used), leading to a total of 240 processors.

#class	4	8	16	32	64	128	256
MemMap Speed-up	-	-	-	-	-	-	-
DMA speed-up	9	16	39	92	169	253	336

SLAAC-1V board

The SLAAC-1V board [10] includes also three Virtex 1000 processing elements with the restriction that processing element X0 is only 50% available for user implementation. The Memory Map transfer rate is equal to 5 Mbytes/sec and the DMA engine provide a bandwidth of approximately 40 Mbytes/sec.

#class	4	8	16	32	64	128	192
MemMap Speed-up	1.8	3.6	7.4	15	30	60	113
DMA speed-up	9	16	39	92	169	253	302

Spyder board

The Spyder board [11] houses a single Virtex 800 component (56 x 84 CLBs) in which a 64 processor array can fit (75% of CLBs used). Tests carried out at IRISA, France, indicate an average Memory Map transfer of 12 Mbytes/sec and a DMA transfer rate of more than 40 M Bytes/sec.

#class	4	8	16	32	64
MemMap Speed-up	3	7	7.4	16	34
DMA speed-up	9	16	39	92	169

7 Conclusion

The speed-up provided by the pixel stream architecture is function of two important parameters:

- the number of classes clustering the image.
- the transfer rate between the host and the FPGA board.

But with the current available FPGA boards such as the SLAAC-1V, the Wildstar or the Spyder boards which house Virtex components and offer (reasonable) DMA transfer, a large number of classes can be processed concurrently without to be slow-down by the data stream rate. Actually, there is a good balance between the DMA transfer rate (around 30-40 Mbytes/second) and the clock frequency we can expect to achieved (30-40 MHz).

The architecture is independent of the size of the image and requires no inboard memory. However, the size of the processor memories must be adjusted according to the number of spectral bands of the hyperspectral image. The resources used by the memory represents actually a large part compared to the total resources of a processor. Tailoring the size of the memory to the application will have an immediate effect of the architecture performance. For instance, resizing the memory to handle 128-bands leads to a multiply the maximum speed-up by a factor 1.5.

The architecture requires no global control, except the clock in the case of a synchronous design such as the systolic array we have presented. The advantage of propagating the control signal along the array is that the clock frequency becomes independent of the size of the array, providing a scalable architecture. In addition, and this may be as important as anything else, the design is greatly simplified: it is mainly reduced to optimized a fairly simple processor.

Acknowledgment

I would like to thank James Theiler for its careful reading and its suggestions to improve this report, and Toni Nelson for the Virtex experiments.

References

[1] M. Lesser, J. Theiler, M. Estlick and J. Szymanski, Design Tradeoffs in a hardware implementation of the k-means algorithm, Proc SAM 2000, First IEEE Sensor Array and Multichannel Signal Processing Workshop, march 2000.

[2] J. Theiler and G. Gisler, A contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation. Proc . SPIE 3159, 1997.

[3] P.M. Kelly and J.M. White, Preprocessing remotely-sensed data for efficient analysis and classification, in SPIE Vol. 1963 Applications of Artificial Intelligence 1993: Knowledge-Based Systems in Aerospace and Industry, pages 24-30, 1993.

[4] Wildforce Reference Manual, revision 3.4, Annapolis Micro System Inc, 1999 (www.annapmicro.com).

[5] Wildstar board, Annapolis Micro Systems, Inc., www.annapmicro.com/PR9126.html

[6] XC4000E and XC4000X Series Field Programmable Gate Arrays, Xilinx Product Specification, May 1999.

[7] Virtex Field Programmable Gate Arrays, Xilinx Product Specification, DS003 (v.2.2), May 2000.

[8] Xilinx Unveils New FPGA Architecture to Enable High-performance, Ten-million system gate Design, Press Backgrounder, may 2000. www.xilinx.com/products/virtex/v2_archbkgr.pdf

[9] Multi-Dimensional Image Processing, DAPS project www.daps.lanl.gov/new/MDIP/chalkmean.html

[10] SLAAC Project, www.east.isi.edu/SLAAC/slide_index.htm

[11] K. Weiss, T. Steckstor, C. Otker, I. Katchan, C. Nitsh, J. Philipp Spyder Virtex X2, User's Manual (v.1.1), sept 1999.

ANNEX 1: Speed-up Calculation

The speed-up is given by the ration $T1/T2$

$$T1 = Tdseq + Tac$$

$$T2 = Tdpar + Tac + Tc_{update}$$

$$Tdseq = cntr_dist \times NB_BANDS \times NB_CLASS \times (1/MEDS)$$

$$Tdpar = cntr_dist \times NB_BANDS \times (1/TR)$$

$$Tc_{update} = cntr_center \times NB_BANDS \times 1/TR$$

Let $\alpha = Tac/Tdseq$ and $Sd = Tdseq/Tdpar$

From experiment results (page 3) we have $cntr_center << cntr_dist$.

The ratio $T1/T2$ can be simplified as follows:

$$\frac{T1}{T2} = \frac{Tdseq + \alpha \times Tdseq}{Tdseq/Sd + \alpha \times Tdseq} = \frac{1+\alpha}{1/Sd + \alpha}$$

Considering $\alpha << 1$

$$Speed-up = S = \frac{T1}{T2} = \frac{Sd}{1 + \alpha \times Sd}$$

Sd is the optimal speed-up provided by the array on the distance computation.

α is a corrector coefficient which takes into account all the other computations, especially the update of the class centers.

ANNEX 2: VHDL code of a systolic processor

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity kmean_process is
 port
   (
     Clk       : in   std_logic;
     NumProc   : in   std_logic_vector (7  downto 0);
     WrMemIn   : in   std_logic;
     WrMemOut  : out  std_logic;
     AddrIn    : in   std_logic_vector (7  downto 0);
     AddrOut   : out  std_logic_vector (7  downto 0);
     PixelIn   : in   std_logic_vector (7  downto 0);
     ClassOut  : out  std_logic_vector (7  downto 0);
     ClassIn   : in   std_logic_vector (7  downto 0);
     PixelOut  : out  std_logic_vector (7  downto 0);
     LdAccIn   : in   std_logic;
     LdAccOut  : out  std_logic;
     MinIn     : in   std_logic_vector (15 downto 0);
     MinOut    : out  std_logic_vector (15 downto 0);
     IdxIn     : in   std_logic_vector (7  downto 0);
     IdxOut    : out  std_logic_vector (7  downto 0)
   );
end kmean_process;

architecture struct of kmean_process is

component mem224x8
  PORT(
    A      : IN std_logic_vector(7 DOWNTO 0);
    DI     : IN std_logic_vector(7 DOWNTO 0);
    WR_EN  : IN std_logic;
    WR_CLK : IN std_logic;
    DO     : OUT std_logic_vector(7 DOWNTO 0));
end component;

component AinfB16
  PORT(
    A      : IN std_logic_vector(15 DOWNTO 0);
    B      : IN std_logic_vector(15 DOWNTO 0);
    A_LT_B : OUT std_logic);
end component;
```

```
component Equ8
  PORT(
    A      : IN std_logic_vector(7 DOWNTO 0);
    B      : IN std_logic_vector(7 DOWNTO 0);
    A_EQ_B : OUT std_logic);
end component;

component Sub16
  PORT(
    A   : IN std_logic_vector(15 DOWNTO 0);
    B   : IN std_logic_vector(15 DOWNTO 0);
    SUM : OUT std_logic_vector(15 DOWNTO 0));
end component;

component Acc16
  PORT(
    ADD_SUB : IN std_logic;
    B       : IN std_logic_vector(15 DOWNTO 0);
    LOAD    : IN std_logic;
    CLK_EN  : IN std_logic;
    CLOCK   : IN std_logic;
    Q_OUT   : OUT std_logic_vector(15 DOWNTO 0));
end component;

signal Pixel    : std_logic_vector(15 DOWNTO 0);
signal MemOut   : std_logic_vector(7  DOWNTO 0);
signal DPixel   : std_logic_vector(15 DOWNTO 0);
signal Center   : std_logic_vector(15 DOWNTO 0);
signal Abb      : std_logic_vector(15 DOWNTO 0);
signal Acc      : std_logic_vector(15 DOWNTO 0);
signal Sub      : std_logic_vector(15 DOWNTO 0);
signal Inf16    : std_logic;
signal equidx   : std_logic;
signal WriteMem : std_logic;
signal Enable   : std_logic := '1';
signal addsub   : std_logic;

begin

PixelOut <= DPixel(7 downto 0);
Pixel  (7 downto 0) <= PixelIn;
Pixel  (15 downto 8) <= "00000000";
Center (15 downto 8) <= "00000000";
WriteMem <= WrMemIn and equidx;
addsub <= not Abb(15);
```

```
process (Sub, LdAccIn)
  begin
    for i in 0 to 15 loop
      Abb(i) <= Sub(i) xor (LdAccIn and Sub(15));
    end loop;
  end process;

process (Clk)
  begin
    if rising_edge(Clk) then
      Center(7 downto 0) <= MemOut;
      DPixel <= Pixel;
      AddrOut  <= AddrIn;
      LdAccOut <= LdAccIn;
      WrMemOut <= WrMemIn;
      ClassOut <= ClassIn;
      if Inf16 = '1'
        then MinOut <= MinIn; idxOut <= IdxIn;
        else MinOut <= Acc;   IdxOut <= NumProc;
      end if;
    end if;
  end process;

memOp0 : mem224x8 port map
(A => AddrIn,
 DI => PixelIn,
 WR_EN => WriteMem,
 WR_CLK => Clk,
 DO => MemOut);

subOp0 : Sub16 port map
(A => DPixel,
 B => Center,
 SUM => sub);

accOp0 : Acc16 port map
(ADD_SUB => addsub,
 B => Abb,
 LOAD => LdAccIn,
 CLK_EN => Enable,
 CLOCK => Clk,
 Q_OUT => Acc);
```

```
infOp0 : AinfB16 port map
(A => MinIn,
 B => Acc,
 A_LT_B => inf16);

equOp0 : Equ8 port map
(A => ClassIn,
 B => NumProc,
 A_EQ_B => equidx);

end struct;
```