Early Experience with a Hybrid Processor: K-Means Clustering

Maya Gokhale, Jan Frigo
Kevin McCabe, James Theiler
NIS-3, NIS-4, NIS-2
Los Alamos National Laboratory
Los Alamos, NM, U.S.A.

Dominique Lavenier
IRISA - CNRS
Campus de Beaulieu
35042 Rennes cedex - FRANCE

Abstract *We discuss hardware/software co-processing on a hybrid processor for a compute- and data-intensive hyper-spectral imaging algorithm, K-Means Clustering. The experiments are performed on the Altera Excalibur board using the soft IP core 32-bit NIOS RISC processor. In our experiments, we compare performance of the sequential algorithm with two different accelerated versions. We consider granularity and synchronization issues when mapping an algorithm to a hybrid processor. Our results show that on the Excalibur NIOS, a 15% speedup can be achieved over the sequential algorithm on images with 8 spectral bands where the pixels are divided into 8 categories. Speedup is limitd by the communication cost of transferring data from external memory through the NIOS processor to the customized circuits. Our results indicate that future hybrid processors must either (1) have a clock rate 10X the speed of the configurable logic circuits or (2) include dual port memories that both the processor and configurable logic can access. If either of these conditions is met, the hybrid processor will show a factor of 10 speedup over the sequential algorithm. Such systems will combine the convenience of conventional processors with the speed of configurable logic.*

Keywords: configurable system on a chip, CSOC, Excalibur, K-means Clustering, image processing

1 Introduction

Over the past ten years, it has been well documented that configurable logic processors composed of SRAM-based Field Programmable Gate Arrays (FPGAs) can accelerate compute-intensive operations by one to two orders of magnitude over Pentium-class processors. However, as more experience has been gained with FPGA processing, it has also become evident that there is much more to any algorithm than a compute-intensive core. File I/O, outer loop management, and other housekeeping tasks make up the bulk of the source code. It is time-consuming to map these functions onto hardware and usually not profitable in terms of speedup - it is better to use hardware to unroll an inner loop for the maximum data flow rather than to map complex control and I/O functions onto hardware.

However, the architecture of currently available FPGA computing platforms does not lend itself easily to hardware/software co-processing. FPGA boards typically communicate with a processor via an I/O bus such as PCI or VME. Not only is the I/O bandwidth between hardware and software slow and pin-limited, but the system overhead to set up a transaction between the processor and FPGA board is high. All these factors dictate that as much of the computation as possible occur in hardware, and that the granularity of transaction between hardware and software is both large and deterministic (so that operations can be scheduled), with minimal synchronization between the two.

Recently, hybrid Configurable System on a Chip (CSOC) architectures, proposed several years ago ([8], [5], [9]), have begun to appear as commercial offerings ([1], [10]). In contrast to traditional FPGAs, these integrated systems offer a processor and an array of configurable logic cells on a single chip. On such systems, it becomes feasible to have software and hardware communicate at clock cycle latency rather than over a slow I/O bus, speeding up synchronization between the two. As a result, a smaller granularity of operation should be possible in hardware as compared to the conventional FPGA board co-processor.

As hybrid processors are still not readily available, there has been to date little experience with mapping algorithms to these devices and measuring performance. In this paper, we present practical experience with using the Excalibur NIOS system for a compute- and data-intensive application in remote sensing, the K-Means Clustering algorithm. We choose this algorithm because it is readily parallelizable in a variety of ways, and FPGA-based acceleration of K-Means kernel loops has previously been reported [7]. We experiment with mapping K-Means to a hybrid processor and evaluate performance of two different mapping techniques.

Figure 1: A Multi-Spectral Image

Assign pixels randomly to NB_CLASS classes
Compute the centers of the classes
Loop(N) For each pixel,

- Let C = class of the pixel
- Determine the class number K which has the minimum distance to C
- if C is not equal to K, move pixel C to Class K

Recompute the centers of the classes K and C

Figure 2: K-Means Clustering Algorithm

2 K-Means Clustering

The basic principle of image clustering is to take an original image and to represent the same image using only a small number of pixel values. The goal of the K-Means algorithm is to assign each pixel to one of a pre-defined number NB_CLASS of classes. The assignment is done by minimizing a cost function over the set of NB_CLASS class centers, where the class center is simply the average of pixel values currently assigned to the class. This iterative algorithm compares each pixel to each class center, finds the class center with minimal distance to the pixel, and then assigns the pixel to that class. The algorithm may be run for a fixed number of iterations or until no pixel has moved to a new class.

The K-Means algorithm is typically applied to multi- and hyper-spectral remote sensing imagery. Figure 1 shows such an image.
In a multi- or hyper-spectral image, each "pixel" is actually a "hyper-pixel," a vector with a component for each spectral channel in the image. A representative hyper-spectral image might contain 512 x 512 hyper-pixels, where each hyper-pixel is a vector of length 224, and each vector component is 8 – 14 bits long.

Figure 2 outlines the K-Means algorithm, and Figure 3 shows the main K-Means loop in C.

A loop iteration scans all the pixels. For each pixel we check if it still 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 in a class is stored as well as the sum ac-

```
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 distance: pixel <=> all classes  */
7        for (k=0; k<NB_CLASS; k++) {
8          if (N_CENTER[k]!=0) {
9            dist = 0;
10           for (d=0; d<NB_BANDS; d++)
11             dist = dist +
12                   ABS (PIXEL[i+b][d] - CENTER[k][d]);
13           /* find min dist and associated class# */
14           if (x<min) { min = dist; idx[b] = k; }
15         }
16       }
17     }
18     for (k=0; k<NB_CLASS; k++) change[k] = false;
19     for (b=0; b<B; b++) {
20       if (CLASS[i+b]!=idx[b]) {
21         pixel_move ++;
22         k = CLASS[i+b]; N_CENTER[k]--;
23         change[k] = true;
24         for (d=0; d<NB_BANDS; d++)
25           ACC[k][d] = ACC[k][d] -
26                       PIXEL[i+b][d];
27         k = idx[b]; CLASS[i+b] = k; N_CENTER[k]++;
28         change[k] = true;
29         for (d=0; d<NB_BANDS; d++)
30           ACC[k][d] = ACC[k][d] + PIXEL[i+b][d];
31       }
32     }
33     for (k=0; k<NB_CLASS; k++)
34      /* recompute centers if needed */
35       if (N_CENTER[k]!=0 && change[k]==true) {
36         for (d=0; d<NB_BANDS; d++)
37          CENTER[k][d] = ACC[k][d]/N_CENTER[k];
38       }
39   }
40 }
```

Figure 3: K-Means C Code

cumulation necessary for recomputing the class centers. In our implementation, the class centers are periodically updated every block of B pixels. The cost function is an approximation described in [2] well suited to our data set and is computed as the absolute value of a difference. This cost function is well suited to today's configurable hardware. In software, the squared difference is usually used.

The computation can be split roughly into three parts: the distance calculation between a pixel and a class center, the accumulator update, and the center update. In [6], we report the results of profiling the K-Means algorithm. We have found that the most time consuming computation is the distance calculation that compares each pixel value to each class center (see lines 3 – 16 in Figure 3). In the case of 32 classes, this loop consumes more than 99.6% of the computation time. Thus, this calculation is the natural candidate for acceleration.

There are many ways to accelerate K-Means on configurable logic. Two different acceleration approaches have been reported in [7] and [6] (see [3] for a summary of[6]). Both methods put the distance calculation (line 11 of Figure 3) in hardware. [7] pre-loads the image into local memory on the FPGA board and performs all computation except the final center mean calculation in hardware. Thus the entire image must fit in local memory. [6] streams the image pixels through board, and performs only the distance calculation in hardware. It can handle arbitrary size images and scales well to a large number of classes. It incurs communication overhead in repeatedly streaming the image from the processor to the hardware.

3 Mapping K-Means onto a Hybrid Processor

Our hybrid processor model is shown in Figure 4. There is a RISC processor with a variable number and size of busses connecting it to configurable logic. The RISC processor and configurable logic share memory. The configurable logic consists of a "sea of gates" along

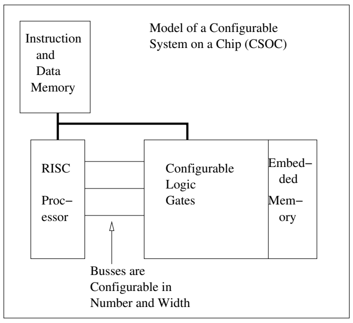

Figure 4: Abstract Hybrid Processor Architecture

with a collection of small embedded memory modules. We refer to a hardware design in the configurable logic as the "user logic." We assume that the processor and user logic run at the same clock speed and that a word may be transmitted between processor and user logic in one clock cycle. The NIOS Excalibur approximates this model, with some important differences. On the NIOS, the user logic cannot access the Instruction and Data SRAM directly. While theoretically the user logic and processor can exchange data in one clock cycle, in reality we measure O(10) clock cycles to send a single 32-bit number from NIOS processor to user logic (see Section 3.1 below).

3.1 Iteration 1: Speeding up Distance Calculation

We approach the problem of mapping K-Means to a hybrid processor incrementally. Since the most time consuming operation is the distance calculation loop, we first map the kernel of that loop to hardware, with all the other code remaining in software. This highlights one of the important advantages of a hybrid processor (see [4] for a more detailed discussion of this point), namely that it is easy with such an architecture to incrementally insert hardware acceleration into a conventional program. We replace a single statement in the C program with a call to the configurable logic. The hardware is a combinational logic circuit with input ports consisting of the distance, the current pixel and current center. The circuit performs the indicated subtraction, abs function and accumulation and returns the updated variable `dist`. Figure 5 shows both the modified C code and the VHDL for this version of the algorithm.[1]

In this example, lines 11 and 12 of Figure 3 are replaced by calls to send the data to the configurable logic and to retrieve the result. The data is sent and received through a set of user-defined busses (see lines 41 – 45).

This hardware logic takes less than 1% of the chip and does not affect the clock frequency of the chip. On the Excalibur, the 32-bit NIOS plus the user logic occupy 22% of the chip, and the clock frequency (fMax) is 31.71. Since we have previously noted that the distance calculation by far dominates the computation time, we might expect the hardware acceleration of this key computation to significantly speed up the K-Means run time. There are two subtracts and one add in the distance calculation. The RISC processor takes at least one clock cycle to execute each of these instructions. All three are done in one clock cycle in the user logic.

In this experiment, the sequential and "accelerated" versions were roughly the same speed. For 64 pixels, with 8 classes and 8 bands, the accelerated version was 15% faster than sequential. When the number of pixels was increased to 224 with 224 classes, the sequential algorithm was 5% faster. This is due to a combination of factors. First, although the arithmetic operations (subtracts and an add) have been accelerated, we have added a cost by communicating the distance, center, and pixel values to the user logic and reading back the updated distance. As the amount of data to be sent to the user logic is increased, the communication overhead begins to dominate the run time.

In an experiment to quantify the cost of sending a single 32-bit value from processor to user logic, we determined that on the Excalibur

[1] Although the send and receive are shown as separate function calls, the calls were manually inlined when measuring speed.

```
Modified C Code:

11 send_data(0,CENTERS,dist,PIXELS);
12 dist = get_result();

...

41 EP_PIO *dist_out = (EP_PIO *) NA_dist_out;
42 EP_PIO *ul_reset = (EP_PIO *) NA_Reset;
43 EP_PIO *center = (EP_PIO *) NA_center;
44 EP_PIO *dist_in = (EP_PIO *) NA_dist_in;
45 EP_PIO *pixel = (EP_PIO *) NA_pixel;
46
47 int send_data(rst, cent, din, pix)
48 int rst, cent, din, pix;
49 {
50   ul_reset->EPR_PIOData = rst;
51   center->EPR_PIOData = cent;
52   dist_in->EPR_PIOData = din;
53   pixel->EPR_PIOData = pix;
54 }
55
56 int get_result()
57 {
58   return (dist_out->EPR_PIOData);
59 }

VHDL for Distance Calculation:

  dist_process: process(Clk, Reset)
    variable p_i: integer;
    variable c_i : integer;
    variable d_i : integer;
  begin
    if (Reset = '1') then
       dist_out <= "0000000000000000";
    elsif rising_edge(Clk) then
        p_i := conv_integer(pixel);
        c_i := conv_integer(center);
        d_i := conv_integer(dist_in);
        if pixel>center then
          dist_out <=
           conv_std_logic_vector(d_i + (p_i - c_i),16);
        else
          dist_out <=
           conv_std_logic_vector(d_i + (c_i - p_i),16);
        end if;
    end if;
  end process;
```

Figure 5: Hardware Acceleration of Distance Calculation

with a 32-bit NIOS processor, it takes 11 clock cycles[2] to send one 32-bit value from processor to user logic using memory-mapped I/O, which is a 12MB/s rate assuming a 33MHz clock for both processor and user logic. This communication cost more than offsets the gain of performing multiple arithmetic operations in parallel. Second, even if we could communicate a word between processor and user logic in a single user logic clock cycle by increasing the processor clock speed by a factor of 10, there is still a significant amount of address calculation code in the innermost loop that is performed sequentially. Thus the fraction of parallel code relative to the amount of sequential code is quite small, which, by Amdahl's Law, is a limiting factor to speedup.

Our conclusion from this experiment is that communication cost continues to be critical to the granularity of the custom instruction. The speed of communication can be increased by increasing the clock speed of the processor relative to the user logic.

3.2 Iteration 2: Parallelizing across Classes

Our second approach focuses on larger granularity with a more global parallelization of the distance calculation by unrolling the loop over all the classes on lines 7–16. The idea is to flow the pixel stream through a linear array of cells, where the number of cells is equal to the number of classes. A cell k computes the distance between its class k and the current flowing pixel. It also updates the current "best" class that has been found for each pixel (i.e., the class with minimum distance to the pixel). The new class computed for the pixel is returned to the processor, and new class centers are computed. Periodically, a new set of centers is streamed to the array of cells. The cell array is shown in Figure 6.

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

[2]This includes 2 wait states.

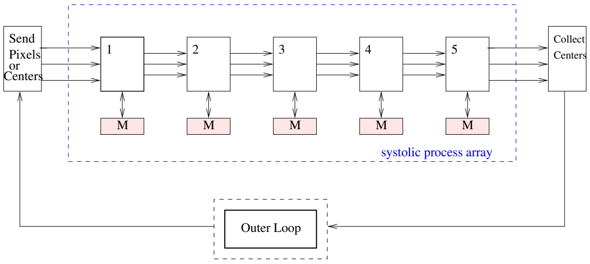

Figure 6: Linear Array Implementation

```
index = my_processor_number;
while (! end_of_stream) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    stream_read (pixel);
    dist = dist + ABS(pixel - center[d]);
    stream_write (pixel);
  }
  stream_read (left_dist, left_index);
  if (dist < left_dist) {
    left_dist = dist; left_index = index;
  }
  stream_write (left_dist, left_index);
}
```

The input data (pixels and class centers) are written from the processor to the user logic through a set of user-defined busses, similar to the method shown in Figure 5.

In this experiment, the accelerated version showed an 11% speedup over the sequential algorithm. Once again, the cost of communicating the image to the user logic, at 11 cycles per word, was the dominating factor that prevented greater speedup. Although there was greater parallel activity in this version than either the sequential version or our first iteration, the high cost of sending the pixel array to the user logic was the limiting factor.

We have compiled this hardware version of the K-Means distance calculation (implemented with 32 classes or cells) for both the Altera APEX20K200 as well as the Xilinx Virtex. The table in Figure 7 shows the size of the user logic on the Altera chip as well two Virtex chips. Each cell holds 224 spectral channels for the class center it represents.

FPGA	Gates	%Usage	Speed
APEX20K200	526K	53%	50 MHz
Virtex V400	468K	70%	25 MHz
Virtex V1000	1.124M	28%	41 MHz

Figure 7: Comparison of K-Means Hardware on Altera and Xilinx (implemented with 32 Classes)

4 Conclusions

We have demonstrated the mapping of a data- and compute-intensive algorithm, K-Means Clustering to a hybrid processor consisting of a RISC processor augmented with configurable logic. We have experimented with two approaches to accelerating the K-Means inner loop with maximum speedup achieved of 15%.

One conclusion we draw from this experiment is that speedup can only be gained by the P-RISC approach of substituting hardware for short segments of sequential instructions if there is very fast communication between processor and user logic. This is especially true if the cost of reconfiguration is factored in, which we did not consider in this experiment.

The higher pay-off approach is to parallelize in hardware at the loop level and to put overhead operations such as address calculation on the hardware. The overhead of communicating data between the processor and user logic remains the primary impediment to higher speedup. Limitations of the clock speed of the soft processor core and memory architecture of the Excalibur NIOS development board are responsible for the measured overhead.

These factors can be remedied by

- using a hard IP core processor that is clocked at 10X the clock rate of the user logic
- separate data and instruction memory
- instruction cache
- dual ported memory shared by processor and configurable logic

- multiple memory banks.

With the dual ported memory, it would be possible to pass to the user logic simply the addresses of the pixel and center arrays, and let the hardware perform pipelined fetch of the pixel data directly. Center update could still be done by the software, with the new class centers written directly to the shared memory. With this approach it is possible for the hybrid chip to deliver one to two orders of magnitude speedup over software.

Extrapolating to the Virtex 1000 PowerPC, we can fit 96 classes. If the memory communication can proceed at 40MB/sec (comparable to 32-bit PCI DMA mode), a speedup of approximately 200 [6] can be realized over the software version. This communication rate can either be achieved by running the processor at 400MHz or by providing a path for the user logic to access memory concurrently with the processor.

Thus, despite the modest measured results from the Excalibur NIOS, we believe it is possible for the performance of a hybrid processor to be orders of magnitude faster than a conventional processor. However, careful attention to the system architecture is necessary to realize these benefits.

Acknowldegements: We are grateful to Konstantin Borozdin for helping compile and debug version 1 of the K-Means to the Excalibur.

References

[1] Altera Corporation. Excalibur. *http://www.altera.com/products/devices/excalibur/exc-index.html*, 2001.

[2] M. Estlick, M. Leeser, J. Szymanski, and J. Theiler. Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware. *ACM FPGA 2001*, 2001.

[3] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. *ACM FPGA 2001*, 2001.

[4] M. B. Gokhale and J. M. Stone. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems*, 24, March 2000.

[5] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, Apr. 1997. To be published.

[6] D. Lavenier. FPGA Implementation of the K-Means Clustering Algorithm for Hyperspectral Images. *Los Alamos National Laboratory LAUR 00-3079*, 2000.

[7] M. Leeser, M. Estlick, N. Kitaryeva, J. Theiler, and J.Szymanski. Applying Reconfigurable Hardware to Segmentation for Multispectral Imagery. In *HPEC 2000*, Boston, MA, Sept. 2000.

[8] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, Nov. 1994.

[9] C. Rupp et al. The Napa Adaptive Processing Architecture. *FCCM 1998*, Apr. 1998.

[10] Xilinx Corporation. Virtex/powerpc. *http://www.xilinx.com/prs_rls/ibmpartner.htm*, 2000.