The Journal of Supercomputing, 26, 131–148, 2003

Experience with a Hybrid Processor: K-Means Clustering

MAYA GOKHALE maya@lanl.gov
Los Alamos National Laboratory, Los Alamos, NM, USA

JAN FRIGO jfrigo@lanl.gov
Los Alamos National Laboratory, Los Alamos, NM, USA

KEVIN MCCABE kmccabe@lanl.gov
Los Alamos National Laboratory, Los Alamos, NM, USA

JAMES THEILER jt@lanl.gov
Los Alamos National Laboratory, Los Alamos, NM, USA

CHRISTOPHE WOLINSKI* krzvsztof.wolinski@irisa.fr
Los Alamos National Laboratory, Los Alamos, NM, USA

DOMINIQUE LAVENIER dominique.lavenier@irisa.fr
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 multispectral imaging algorithm, k-means clustering. The experiments are performed on two models of the Altera Excalibur board, the first using the soft IP core 32-bit NIOS 1.1 RISC processor, and the second with the hard IP core ARM processor. In our experiments, we compare performance of the sequential k-means algorithm with three different accelerated versions. We consider granularity and synchronization issues when mapping an algorithm to a hybrid processor. Our results show that speedup of 11.8X is achieved by migrating computation to the Excalibur ARM hardware/software as compared to software only on a Gigahertz Pentium III. Speedup on the Excalibur NIOS is limited by the communication cost of transferring data from external memory through the processor to the customized circuits. This limitation is overcome on the Excalibur ARM, in which dual-port memories, accessible to both the processor and configurable logic, have the biggest performance impact of all the techniques studied.

Keywords: configurable system on a chip, CSOC, Excalibur, FPGA, 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 certain classes of compute-intensive operations by one to two orders of

*Also at IRISA–CNRS, Campus de Beaulieu, 35042 Rennes cedex, France.

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 house-keeping 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.

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 [7, 12, 13], have begun to appear as commercial offerings [1, 18]. 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-processer.

As hybrid processers 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 Excaliber NIOS 1.1 and ARM systems for a computer- and data-intensive application in remote sensing, the k-means clustering algorithm. We chose 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 [10]. We experiment with mapping k-means to two hybrid processors and evaluate the performance of three different mapping techniques.

2. K-means Clustering of Multi- and Hyper-Spectral Imagery

2.1. Algorithm Overview

In a multi- or hyper-spectral image (See Figure 1), 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 512×512 hyper-pixels, where each hyper-pixel is a vector of length 224, and each vector component is $8-14$ bits long.

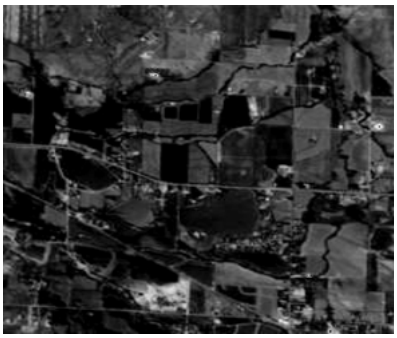

Figure 1. One channel of a multi-spectral image.

In general, k-means produces a partition of the pixels in a multi-spectral image into distinct classes in such a way that two pixels in the same class are spectrally similar. A map of distinct classes is a data product with a number of applications, including quicklook generation, data compression [11], image restoration [15], remote sensing change detection [14], and clutter reduction for weak signal detection [5].

The standard algorithm is quite straightforward. Each iteration consists of a single pass over every pixel in the data set; during this pass, distances are computed from each pixel to each of the K centers. The pixel is assigned to the cluster to whose center it is closest. At the end of the pass, the centers are recalculated from the new assignments. If after an iteration, none of the pixels are reassigned to new classes (so that the centers do not change), then convergence is complete. Each iteration of the k-means algorithm can be shown to reduce the in-class variance, but the final converged solution is a local, and not necessarily a global, minimum.

A large number of variants have been considered, of which we will discuss two. The first is a ''block'' k-means (e.g., see [17]). In this variant, a full iteration consists of a number of passes over blocks of pixels at a time. Rather than wait until the end of the full iteration to update centers, the centers are updated at the end of each block. Since this gives the centers more opportunities to ''migrate to stable positions, the block k-means algorithm often achieves convergence in fewer iterations than the standard approach. Standard k-means is the same as block k-means with block size equal to the number of pixels in the image.

A second variant is a hierarchical approach, and is based on the observation that convergence is usually much faster if there are fewer classes. In this variant, one begins with a small number of classes (usually 2), and then performs one or two iterations. One then splits each cluster into two by adding a small perturbation to the center positions. This process is continued until the desired number of classes has been obtained, and after that, the ordinary k-means continues until convergence. Note that this second variant can employ either standard or block k-means iterations. The classic Linde-Buzo-Gray algorithm [11] for vector quanization is a hierarchical k-means.

Both of these variants entail a fair amount of bookkeeping, compared to the standard algorithm, and the variants also introduce new free parameters. Coding

these more sophisticated algorithms directly in hardware would take a considerable effort in hardware design. But a well planned software/hardware co-design will allow the hardware to drive high-performance fast iterations, while the software can take care of bookkeeping details that might lead to more intelligent convergence, either by requiring fewer iterations, or by converging to higher-quality solutions [16]. For instance, more sophisicated variants of the hierarchical scheme that involve iterative splitting and merging [8] have been shown to produce even higher quality clustering. Although we did not implement the splitting and merging, to do so would only involve modification of the software—the hardware design would be exactly the same.

To illustrate the benefits of these variants, a software-only simulation was performed on an artifical data set with 5,120 pixels and three bands; the task was to cluster these pixels into 256 distinct classes. These numbers correspond to a common problem in image processing: given a 24-bit image with 8-bit red, green, and blue planes, find the "best" 256-color palette that allows the image to be displayed with 8-bit pixels. Here the cluster centers would correspond to the RGB values of the colors in the palette. The results shown in Figure 2 correspond to ten separate trials (using different random number seeds) of the three variants: the first is standard k-means, the second is block k-means, and the third is hierarchical k-means with blocks. In-class variance is plotted against iteration number, and the curves end when the iterations have completed. Block k-means is seen to achieve high-quality solutions much more rapidly than standard k-means, and achieves final convergence in roughly half the iterations. Hierarchical k-means converges more slowly at first, but it still reaches its final convergence in about the same number of iterations as block k-means. It also achieves higher-quality solutions than either block or standard k-means, and it achieves those better solutions well before its final convergence.

In the experiments that we describe later in Section 4.2, variants of the block update technique have been implemented. Because of limitations (limitations of memory, primarily) in these early generations of CSOC, the parameters we employ in the hardware involve fewer pixels and fewer classes (but more channels) than the data sets used in the experiments above. The hybrid processor approach allows us to

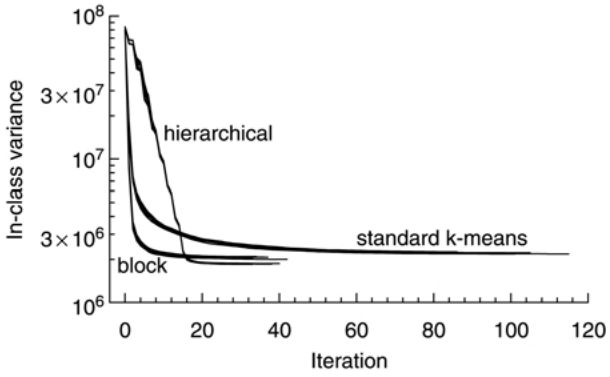

Figure 2. Performance of three different variants of the k-means algorithm.

experiment with software parameters and techniques, while using the same hardware design.

2.2 K-means implementation

Figure 3 shows the C source code for the main k-means loop. 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 accumulation necessary for recomputing the class centers. In our implementation,

```
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 (dist <= 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.

the class centers are periodically updated every block of B pixels. The distance measure cost function is an approximation described in Estlick et al. [3] well suited to our data set and computed as the absolute value of the difference between pixel element and center element. 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 Lavenier [9], 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 3–17 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 Lavenier [9] and Leeser et al. [10] (see Frigo et al. [4] for a summary of Lavenier [9]). Both methods put the distance calculation (lines 11 and 12 of Figure 3) in hardware. Leeser et al. [10] 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. Lavenier [9] streams the image pixels through the 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. Hybrid processor: Model and realizations

An abstract model of a hybrid processor 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 with a collection of small embedded

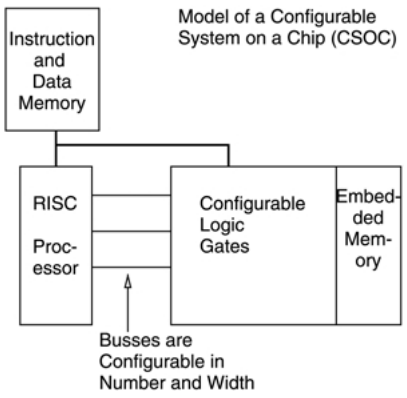

Figure 4. Abstract hybrid processor architecture.

memory modules. We refer to a hardware design in the configurable logic as the "user logic." For some architectures such as the ARM, the processor and user logic execute in different clock domains, so that synchronization is required for direct communication between the two. The NIOS processor and user logic share the same clock.

The NIOS soft core embedded processor fits on the APEX20K programmable logic device (PLD). It is a general-purpose, pipelined, single-issue RISC processor core which processes instructions every clock cycle. The processor interface consists of a user-defined address map with different types, widths and speeds of memory and peripherals. A peripheral bus module (PBM) is the logic interface between the embedded processor and the user logic. This bus module is generated by the NIOS tools according to the user configuration specified (see Figures 5 and 6). Peripheral interfaces include a UART, timer, SRAM, FLASH, user-defined Parallel Input/Output (PIO), and Memory-mapped input/output ports (see Figure 6). The NIOS 1.1 processor with a 32-bit data path configuration utilizes 20% of the available logic elements on the APEX20K200E (1,700 logic elements). We have implemented a user-defined PIO (Section 4.1) and memory-mapped I/O ports for communication from the processor to the user logic.

The NIOS Excalibur approximates the abstract model, with some important differences. On the NIOS, the user logic cannot access the Instruction and Data SRAM directly. Since the NIOS is a soft IP core, there is a single clock controlling both processor and user logic. As the NIOS design is heavily pipelined, a RISC instruction can execute every clock cycle (in the absence of branches). However, we measure O(10) clock cycles to send a single 32-bit number from the NIOS processor to the user logic due to address generation on the processor, and a multi-cycle bus transaction (see Section 4.1 below).

The ARM-based hard core embedded processor fits on an APEX20K PLD. In addition to the processor, the EPXA10 has 38,400 logic elements or 1 million gates. The PLD architecture consists of an embedded processor bus structure, on-chip memory, and peripherals. Figure 7 shows the ARM structure. The embedded processor stripe contains the ARM processor core, peripherals, and memory subsystem. Our system has 256 KB and 128 KB of single- and dual-port memory respectively.

The ARM922T processor core is implemented using a five-stage pipeline. This implementation allows single clock-cycle instruction operation through simultaneous fetch, decode, execute, memory, and write stages. It supports both the 32-bit ARM and 16-bit Thumb instruction sets [2]. Two AMBA-compatible AHB buses serve the ARM-based embedded processor. Three bidirectional AHB bridges enable the peripherals and PLD to exchange data with the ARM embedded processor.

The PLD can be configured via the configuration interface or the embedded processor to implement various devices such as: a master and/or slave peripheral that connects to the embedded bus; on-chip and off-chip memories sharing the stripe; standard interface to on-chip dual-port RAM (allowing SRAM to function as a "large" embedded system block (ESB)).

The master/slave-memory ports are synchronous to the separate PLD clock domains that drive them. The embedded processor domain and PLD domain can be

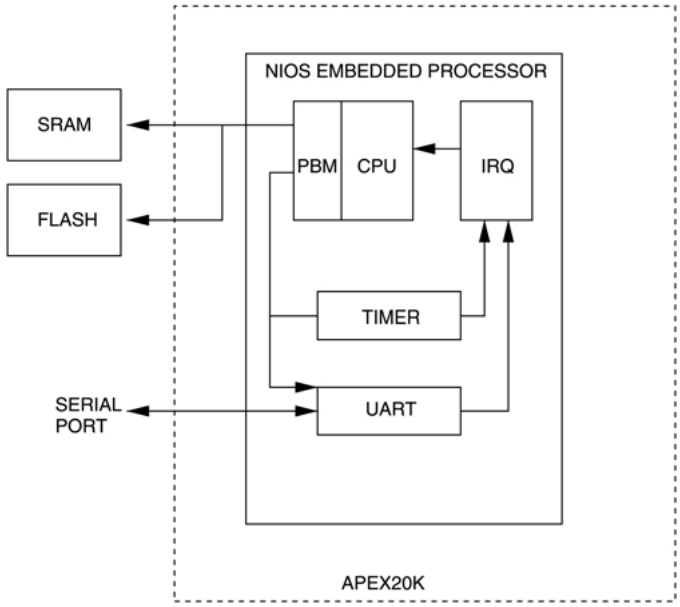

Figure 5. NIOS processor architecture.

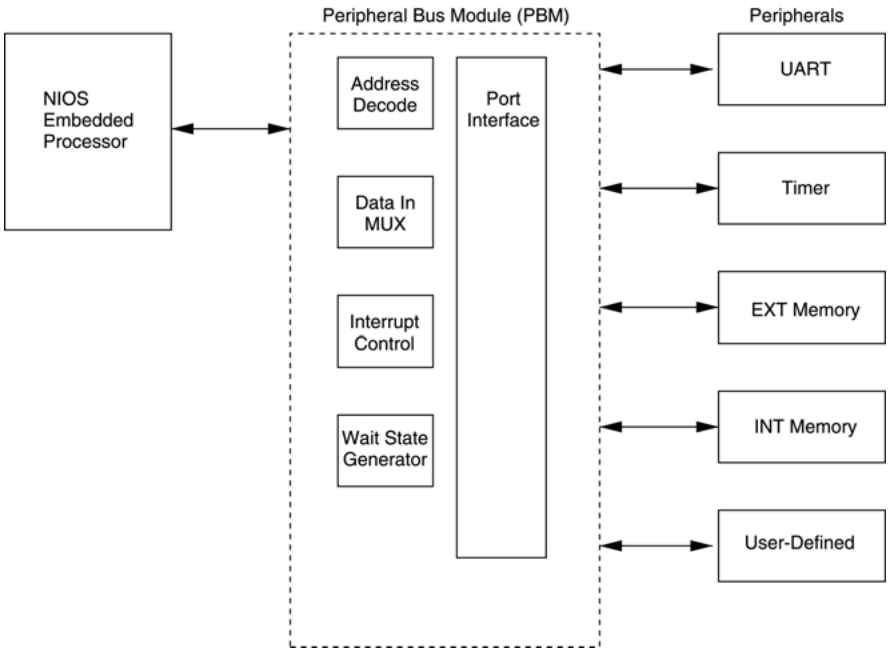

Figure 6. NIOS processor peripherals interface.

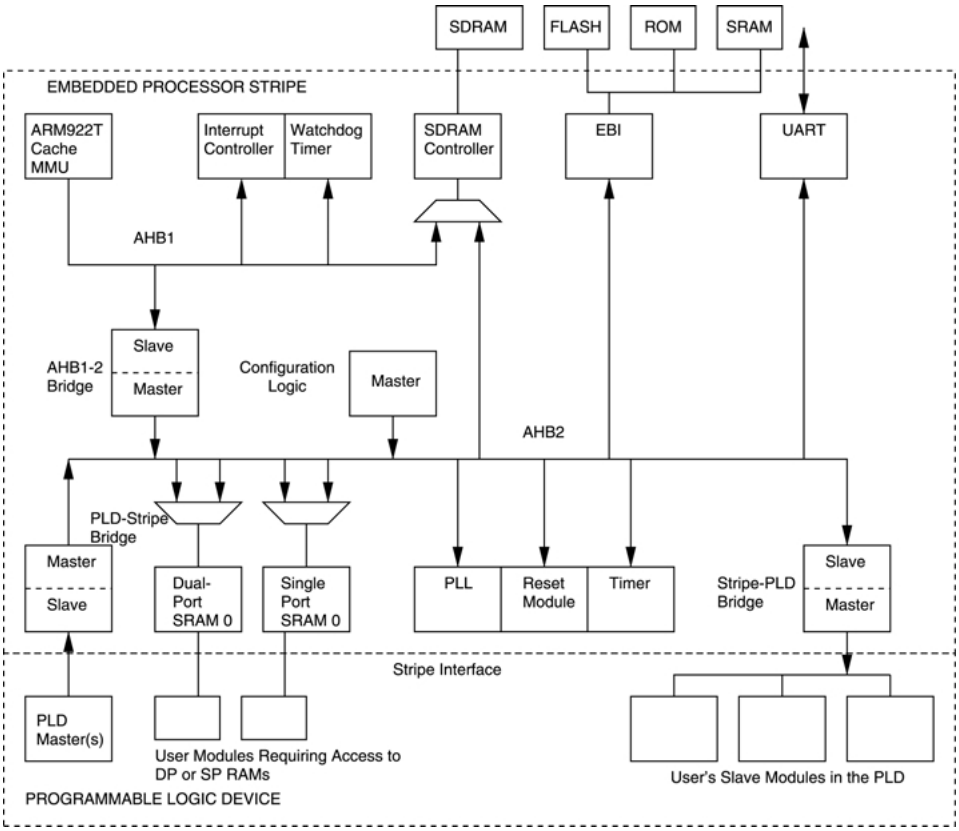

Figure 7. ARM processor architecture.

asynchronous, to allow separate/optimized clock frequencies for each domain. Resynchronization across the domains is handled by the AHB bridges within the stripe. Both the master port and slave port of the stripe are capable of supporting 32-bit data accesses to the whole 4-Gbyte address range (32-bit address bus).

Our application needs high data rate communication between the processor and user logic which is realized in two ways: using the AHB buses/bridges (Section 4.2); and the user-defined configuration interface to on-chip dual-port RAM (Section 4.3).

4. Mapping k-means onto a hybrid processor

4.1. Iteration 1: Speeding up the 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

```
Modified C Code:

11  host2user->np_piodata = (PIXELS[i+b] [d]) + (CENTERS[k] [d]<<8) + (dist<<16);
12  dist = user2host->np_piodata;

41  np_pio *ul_reset = (np_pio*) na_u_reset;
42  np_pio *host2user = (np_pio*) na_host2user; /*PIO bus*/
43  np_pio *user2host = (np_pio*) na_user2host; /*PIO bus*/

VHDL for Distance Calculation:

  Pix <= Host2UlogicData(7 downto 0);
  Center <= Host2UlogicData(15 downto 8);
  DistIn <= Host2UlogicData(31 downto 16);

 dist_process: process(Clk,Reset)
begin
if (Reset = '1')then
  Ulogic2HostData <= x"00000000";
elsif rising_edge(Clk) then
 if (conv_integer(Pix) > conv_integer(Center)) then
    DistOut <= DistIn + (Pix - Center);
 else
    DistOut <= DistIn + (Center - Pix);
 end if;
 Ulogic2HostData <= x"0000" & DistOut;
end if;
```

Figure 8. Hardware acceleration of distance calculation.

software. This highlights one of the important advantages of a hybrid processor (see Gokhale and Stone [6] for a more detailed discussion of this point), namely that it is easy with such an architecture to incrementally insert hardware acceleration into a software program. We replace a single statement in the C program (line 11 and 12 from Figure 3) with a write to and a read from the configurable logic. The hardware is a combinational logic circuit with a 32-bit input register 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 8 shows both the modified C code and the VHDL for this version of the algorithm.

In this example, lines 11 and 12 of Figure 3 are replaced by a write to a parallel I/O port to send the data to the configurable logic and a read from another parallel I/O port to retrieve the result. The data is sent and received through a set of user-defined busses (see lines 42 and 43 in Figure 8).

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 processor plus the user logic occupy 49% of the logic elements on the chip at a clock frequency (fMax) of 34.51 MHz. 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 software-only and hardware-assisted versions were measured for speed with respect to the loop over B pixels. For 64 pixels, with 8 classes and 8 bands, the B loop for the hardware-assisted version was 50% slower than the sequential. This result is due to a combination of factors. First, although the arithmetic operations (subtracts and an add) have been accelerated in hardware, we have added a cost by communicating the distance, center, and pixel values to the user logic and reading back the updated distance. Measuring the distance calculation only, the hardware-assisted version takes 35 cycles verses 25 cycles for the software-only. As the amount of data to be sent to the user logic is increased, the communication overhead dominates 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 with a 32-bit NIOS processor, it takes 11 clock cycles[1] to send one 32-bit value from processor to user logic using memory-mapped I/O or parallel I/O ports, which is a 12 MB/s rate assuming a 33 MHz 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.

This result—a slowdown when the distance calculation kernel is mapped to hardware—is also observed on the Excalibur ARM. Measuring this distance calculation on the ARM, we found the sequential version took 0.08 us compared to 0.4 μs for the "accelerated" version. The cost of communicating through multiple busses and bridges yields a 5 $\times$ slow down in speed. Our conclusion from this experiment is that communication cost continues to be critical to determining the granularity of the custom instruction. Co-locating the processor and user logic on a single device is no guarantee that arbitrarily fine granularity operations mapped onto user logic will improve performance.

4.2. Iteration 2: Parallelizing across classes

Our second approach focuses on increasing the granularity of operation mapped to hardware. By unrolling the loop over all classes on lines 7–16 of Figure 3, we can compute all distances in parallel. The idea is to flow the pixel stream through a linear array of cells, where each cell corresponds to a class. Each cell holds the center for its class in local memory. The cell computes the distance between its class and the current flowing pixel, and 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 by the processor. Periodically, a new set of centers is streamed to the array

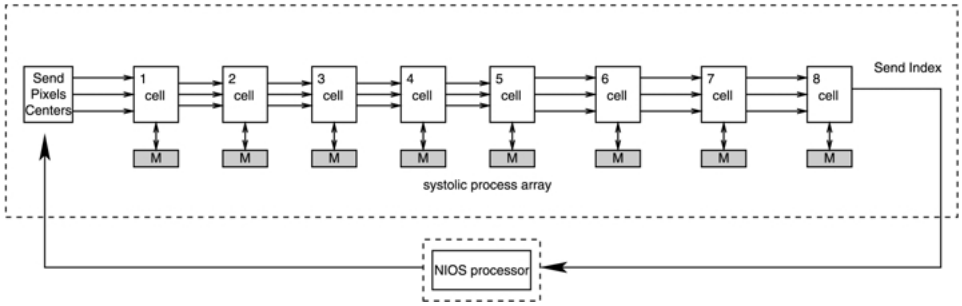

Figure 9. Linear array implementation.

of cells. The cell array (for eight classes) is shown in Figure 9, and the computation each cell performs is shown in Figure 10.

The algorithm operates as follows: First, the centers are sent to the cell array in user logic, with each cell storing the center associated with its class in local memory. The cell array receives a block of pixels, and computes the minimum distance for each class, returning, for each pixel in the block, the index of the class for which a minimum distance is found. The pixel vector has eight bands, with four 8-bit bands packed into one 32-bit word. Two writes to user logic are necessary to send one pixel. The input data (pixels and class centers) are sent to the user logic through a user-defined memory-mapped I/O bus, `*ram`. The associated C code is shown in Figure 11, replacing lines 7–16 in Figure 3.

The software uses a vector of indices, `idx[B]` to recompute the centers, then sends the updated centers back to the user logic. The algorithm continues evaluating another block of B pixels, and updating centers until NB_PIXELS have been evaluated. The algorithm iterates until no more pixels change their association with a class.

As in Iteration 1, the software-only k-means algorithm runs on the NIOS 1.1 processor. The accelerated k-means version runs on both the NIOS 1.1 processor

```
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);
}
```

Figure 10. Hardware acceleration of unrolled B loop.

```
/* Send Centers to user logic */
for (k = 0; k < NB_CLASSES; k++)
{
  *ram = (FLAG_CENTER + (k << 8) + (1 << 16)); /* send center flag, class no. = k */
  for (d = 0; d < NB_BANDS; d = d + 4)
    {
       *ram = CENTERS[k][d] + (CENTERS[k][d+1]<<8) +
              (CENTERS[k][d+2] << 16) + (CENTERS[k][d+3] << 24);
    }
}
 ....

int *ram = (int*) na_user; /* memory mapped nios user logic register */

union packed_pixel {
  int seqpixel; //sequential code, pixel data(8 bits)
  unsigned char upixel[4]; //ulogic, pixel data(32 bits)
} ulogic_pixels[NB_PIXELS] [(NB_BANDS/4)];


*ram = FLAG_PIXELS + (B << 16); /* control word = send pixels to user logic */
for (b = 0; b < B; b++) {  /* for B pixels */
  for (d = 0; d < 2; d++) {  /* for NB_BANDS */
*ram = ulogic_pixels[i+b][d].seqpixel; /* Send 4 8-bit bands to user logic */
  }
  idx[b] = *ram; /* read the index from user logic */
}
```

Figure 11. C code for accelerated k-means version.

and the user logic gates of the APEX20K. In this experiment we timed the number of cycles required to send updated centers to the hardware (center update loop), the distance calculation loop (B loop), and the total number of cycles to complete the entire program. As shown in Figure 12, the accelerated version has 25× speedup in the distance calculation loop and an over all 6× speedup over the software-only version.

Motivated by the analysis described in Section 2.1, we studied performance of the accelerated version with differing block sizes. The results shown in Figure 12 are with block size 1, so that $B = 64 = \text{NB\_PIXELS}$. By varying the block size, we can measure the effect of center update (copying the new center values to the user logic) on overall runtime. Updating the user logic centers takes 846 cycles every B iterations. As B increases, the number of times in the center update loop decreases. Also, as B increases, more pixels are sent to the user logic at a time and the number of times the software loop instructions must be executed decreases. We found for $B = 1$, the distance loop speedup is 13×. For $B = 4$, the distance loop speedup is 20×. For $B = 64$, as seen in Figure 12, the speedup is 25×.

	Sequential cycles	Time (μs)	Accel cycles	Time (μs)	Speed up
Center Update loop	0	0	846	26	NA
Send B Pixels loop	598,101	18,124	23,790	721	25X
Total Algorithm	687,313	20,828	112,443	3,407	6X

Figure 12. Results for NIOS iteration 2 ($B = 64$).

As predicted in Section 2.1 our results show that the block k-means algorithm achieves convergence in fewer iterations than the standard approach, i.e., $B = 64 = \text{NB\_PIXELS}$. (The block k-means converged in four passes while the standard k-means converged in six passes.) However, overall runtime is faster for standard k-means as described above. Also, by altering the software (but not the hardware design), we were able to implement the hierarchical k-means algorithm described in Section 2.1. While simulations show this approach can lead to faster convergence and/or higher quality solutions in some situations (*e.g.*, see Figure 2), we did not expect (or observe) actual speedup for the parameters in our system (NB_PIXELS = 64).

This algorithm was also mapped to the ARM system. In comparison to the NIOS, the ARM architecture employes a hard core RISC processor at 200 MHz with two busses at 200 MHz and 100 MHz respectively. The 100 MHz bus connects the processor to the PLD, APEX20K (see Figure 7). Due to differences in clock speed and communication between NIOS and ARM, the user logic design for the ARM had to be slightly modified to synchronize the bus communication with the ARM. Explicit synchronization was not required on the NIOS for reading and writing data to the user logic because the communication time to send and receive data is slower than the speed of the processor and user logic. For the ARM, bus synchonization requires the use of a *read ready* signal and a *write ready* signal from the user logic. The hardware design took 0.05% of the user logic elements on the PLD at a speed 33 MHz. With the 200 MHz ARM clock, the processor was idle much of the time, waiting for the hardware to complete. As with the NIOS, we used parameters $\text{NB\_PIXELS} = B = 64$. The timing results are shown in Figure 13. Here we see a much more modest speedup over the software-only version. This is principally due to the fact that the NIOS runs at 33 MHz whereas the ARM runs at 200 MHz. The ARM software-only version is inherently faster than the NIOS, while the user logic stays about the same on both.

4.3. *Iteration 3: Exploiting dual port memory*

We demonstrate the advantage of a dual ported memory between the processor and user logic with a new k-means co-design implemented on the Excalibur ARM hybrid processor. This feature is not easily available on the NIOS. For our k-means experiment the processor passes to the dual ported memory two arrays corresponding to the class centers, and a block of B pixels. The processor reads back an array of B results. A result is an index of the class which gives the smallest distance between

	Sequential cycles	Time (μs)	Accel cycles	Time (μs)	Speed up
Center Update loop	0	0	1,482	14.8	NA
Send B Pixels loop	62,754	627	14,926	149	4X
Total Algorithm	406,982	4,069	128,006	1,280	3X

Figure 13. Results for ARM iteration 2 ($B = 64$).

the pixel and its center. As shown in Figure 14, a separate bridge (AHB) between the processor and user logic is used to configure the user logic dual-port access controller, and to command and synchronize the user logic to the processor. The ARM processor can only issue a command to the user logic when the user logic is ready.

In our implementation there are three commands:

- The first command defines the base address of the center array in the dual ported memory along with the number of spectral bands and the number of classes. The user logic loads the class centers when this command is received.
- The second command specifies the base address of the pixel array, the number of spectral bands, and the number of 32-bit words to read. The latter is NB_PIXELS/4, as the hardware is designed to process four pixels every clock cycle. The user logic processes the distance calculation when this command is received.
- The third command specifies the base address of the results array for the processor. The user logic stores the results at this location. In our implementation this command is only sent once. The results overwrite the indexes previously calculated. It is possible to speed up the calculation by pipelining the center update (performed by the ARM) while the user logic computes new indices. In this case we would need to send a different result array address at the beginning of each iteration.

A new hardware design was created so that data could be consumed at the memory access rate. In contrast to the systolic design of Iteration 2, in this design the distance calculation is executed in parallel by an array of 32 processing elements organized as four rows of eight processors. There is one row for each pixel and eight

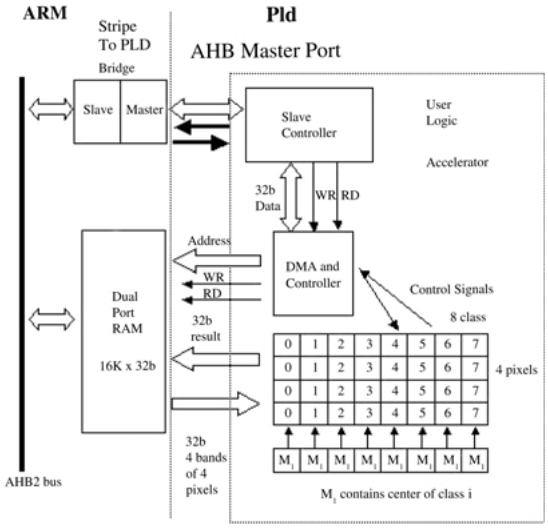

Figure 14. Dual port memory implementation.

processors in the row, one for each class. The distance matrix is stored in a transposed format, with each 32-bit word containing a spectral band from each of four pixels.

The processing elements receive pixel data from the dual port memories and class centers from eight separate memories M_i connected to each column. The user logic processing rate matches the user logic memory access rate of one transaction per clock cycle. The user logic is a two stage pipeline, the first stage computes the distance and the second stage computes the class indexes that correspond to the minimum distance between the pixels and the class centers. The dual-port memory is organized as 16K × 32 bit words, where each word contains four 8 bit spectral bands organized as four spectral bands from four different pixels.

The processing time for Iteration 3 is $(B/4 * (\max(\text{NB\_BANDS} + \texttt{coef1} + \texttt{coef2}), \text{NB\_CLASS} - 1) + \text{NB\_CLASS} + 1$ cycles where the coefficient $\texttt{coef1} = 1$ and $\texttt{coef2} = 1$. $\texttt{coef1}$ can be reduced to 0 if the dual port memory is an asynchronous memory instead of a synchronous one that has a one clock latency between address and data. `coef2` is introduced by pipelining in the processing element between the absolute value calculation and accumulation. Results shown in Figure 15 are for NB_CLASSES = 8, NB_BANDS = 8, $B = 8$, 8-bit pixel data.

Theoretically, at 33 MHz the user logic needs about 0.87 microseconds to calculate the indices for eight pixels composed of eight bands and eight classes. Figure 15 shows a measured time of 1.3 μs, for completion of the synchonization and calculations in the user logic. (Synchronization time is the difference of these times, 0.43 μs.) Iteration 2 user logic needs 19 μs for the same calculations, giving a speedup of 14.7 × over Iteration 2 in the distance loop and 60 × speedup over software-only (labeled "Seq." in Figure 15). Acceleration is still limited by the 32-bit communication bandwidth between the processor and the user logic. The ARM is connected to the dual port memory by a bridge and access by the ARM is eight times slower than access to the ARM's local memory. We note that it is desirable to have a wide data path from the user logic to memory and to have the memory working in asynchronous mode in order to speed up processing.

Finally, we compare the speed of this design to a conventional host computer such as a 1 GHz Pentium-III. For NB_PIXELS = $B = 64$, we measured 65 μs to perform one B loop iteration of the algorithm, i.e., distance calculation loop. The center update can take from 2 μs to 20 μs, depending on the number of centers that need to be updated. For a run with many iterations, only the early iterations will take the longer time; thus the time required for B loop iterations is dominant. The user logic theoretical time for 64 pixels, 8 bands and 8 classes at 33 MHz is: $((64/4)(8+2)+8+1)/33\,\text{Mhz} = (16 * 10 + 9)/33 = 169/33 = 5.1\,\mu\text{s}$. If we take into account synchronization time, 0.43 μs, the total time is the sum of theoretical and

	Sequential cycles	Time (μs)	Iteration 2 cycles	Time (μs)	Iteration 3 cycles	Time (μs)	Speed up over seq.	Speed up over Iter. 2
Send Centers loop	0	0	1,482	14.8	246	2.4	NA	6x
Send B Pixels loop	7,867	78.6	1,925	19.2	138	1.3	60.5X	14.7X

Figure 15. Results for ARM accelerated version 2 ($B = 8$).

synchronization time, 5.53 μs. Thus, we realize a speed up of 11.8× over the 1 GHz machine. Note, the random data generated for the the NB_PIXELS array is identical in all the numerical results presented. If different data sets are used, with less "randomness" in the data, the algorithm will generally converge in fewer iterations, but the B-loop time speed should not vary.

5. 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 three approaches to accelerating the k-means inner loop with maximum speedup achieved of 11.8× over a GHz machine.

One conclusion we draw from these experiments is that speedup can only be gained by the Programmable-RISC approach [12] 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. For small granularity custom instructions (less than 100 RISC instruction), speedup is not realizable on the Excalibur architecture, and the reverse effect may occur.

The higher pay-off approach is to parallelize algorithmic loops in hardware and to put overhead operations such as address calculation on the processor. Even with this approach, the overhead of communicating data between the processor and user logic remains the primary impediment to higher speedup. This limitation is overcome by using dual port memory so that the software can pass to the user logic the addresses of the data and result arrays, and the user logic can access those arrays asynchronously. For the k-means algorithm, we can pass the pixel and center array addresses, and let the hardware perform pipelined fetch of the pixel data directly. Software then reads back the updated centers for each pixel. Using dual-port memory in this way, it should be possible for the hybrid hardware/software to deliver an order of magnitude speedup over high performance software-only.

Acknowledgment

We are grateful to Konstantin Borozdin for helping compile and debug version 1 of the k-means to the Excalibur.

Note

1. This includes 2 wait states.

References

1. Altera Corporation. Excalibur. *http://www.altera.com/products/devices/excalibur/exc-index.html*, 2001.
2. Altera Corporation. ARM-based embedded processor device overview data sheet. *http://www.altera.com/literature/lit-exc.html*, Feb. 2001.
3. 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.
4. J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-C C-to-FPGA compiler: An applications perspective. *ACM FPGA 2001*, 2001.
5. C. Funk, J. Theiler, D. A. Roberts, and C. C. Borel. Clustering to improve matched-filter detection of weak gas plumes in hyperspectral imagery. *IEEE Trans. Geosci. Remote Sensing*, 39:1410–1419, 2001.
6. M. B. Gokhale and J. M. Stone. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems*, 24: March 2000.
7. J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, eds., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, Apr. 1997.
8. T. Kaukoranta, P. Fränti, and O. Nevalainen. Iterative split-and-merge algorithm for vector quantization codebook generation. *Opt. Eng.*, 37:2726–2732, 1998.
9. D. Lavenier. FPGA implementation of the k-means clustering algorithm for hyperspectral images. *Los Alamos National Laboratory LAUR 00-3079*, 2000.
10. 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.
11. Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Trans. Communications*, COM-28:84–95, 1980.
12. 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*, pp. 172–180. IEEE/ACM, Nov. 1994.
13. C. Rupp, et al. The Napa Adaptive Processing Architecture. *FCCM 1998*, Apr. 1998.
14. R. A. Schowengerdt. *Techniques for Image Processing and Classification in Remote Sensing*, Academic Press, Orlando, 1983.
15. D. G. Sheppard, A. Bilgin, M. S. Nadar, B. R. Hunt, and M. W. Marcellin. Vector quantizer for image restoration. *IEEE Trans. Image Processing*, 7:119–124, 1998.
16. J. Theiler, J. Frigo, M. Gokhale, and J. J. Szymanski. Co-design of software and hardware to implement remote sensing algorithms. *Proc. SPIE*, 4480, 2001.
17. B. Thiesson, C. Meek, and D. Heckerman. Accelerating EM for large databases. Technical Report MSR-TR-99-31, Microsoft Research, Microsoft Corporation, Redmond, WA 98052, 1999.
18. Xilinx Corporation. Virtex/powerpc. *http://www.xilinx.com/prs_rls/ibmpartner.htm*, 2000.