Fast Region Labeling on the Reconfigurable Platform ACE-V

Christian Schmidt and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23, D-38106 Braunschweig, Germany
schmidt,koch@eis.cs.tu-bs.de

Abstract. We present a refinement of our previous work on using a reconfigurable computer for quickly labeling connected regions in black and white images. The performance of this solution easily surpasses a state of the art RISC processor and is extremely competitive even with regard to a modern CISC processor. Furthermore, the current hardware implementation has considerable head room for further performance improvements both on the architecture as well as device levels.

1 Introduction

Configurable computers have enjoyed many successes in diverse application areas. From ASIC emulation over gene sequence matching to cryptography and classical signal processing tasks, significant gains in performance have been demonstrated. Another algorithm that has been successfully implemented in the past is labeling connected regions in images. This is a task commonly required in computer vision applications, e.g., for robotics control purposes.

In this paper, we will re-examine this application, and compare results achievable on current architectures with previous work both by our group [1] [2] as well as one of the first attempts targeting the pioneering Splash-2 custom computing machine [3].

2 Algorithm Basics

The algorithm used in our new implementation is fundamentally similar to that of the previous version [1] [2], which in turn was based on [4]. However, some limited changes were being made to increase the performance and better exploit the architectural capabilities of our current hardware platform (see Section 3).

The algorithm expects as input a black and white image organized as a two-dimensional bit array (Figure 1). The value '1' marks a set foreground pixel, '0' indicates a blank background pixel. The output is a two-dimensional array of 16b words having the same dimensions as the input array, but now labeling all pixels within the same individual object with a unique integer identifier.

The original version required the input image to also consist of 16b words, with only the LSB actually being used. By moving to the current denser representation and adding an unpacking phase in hardware, we can now significantly reduce data transfer

Figure 1. Sample input image

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

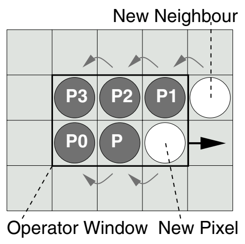
overhead. For purposes of this explanation, assume that the binary image has already been unpacked into 16b words.

In addition to the image itself, we employ two data structures: An *adjacency map* m indexed by label describes the adjacency relationships between label values. E.g., the expression $m[2] = 3$ indicates that pixels which have been labeled with the object ID 2 are adjacent to pixels that have been labeled with the ID 3. In Phase 3 of the algorithm, these relationships will be transitively flattened to only use a single label within the same object.

In some cases, data already entered into adjacency map may become invalid when additional adjacency relations are discovered later in the image. In these cases, the map needs to be updated. For performance reasons (reducing pressure on the memory holding the map), such updates are deferred until the core algorithm itself does not require access to the adjacency map (e.g., during runs of ‘0’ pixels). In the meantime, the adjacency corrections are stored in a separate data structure, called the *adjacency list*, as pairs (a,b) of 16b integer labels. When the map data structure does become available, entries in the list are removed and used to correct the corresponding map entries. The adjacency list is a refinement over the approach we used previously in [1][2].

The labeling procedure itself consists of three distinct phases, with the first phase performing two operations in parallel.

Figure 2. Operator window



1. **Pixel labeling**. Here, we aim to label *adjacent* pixels by assigning them the same object ID. To this end, we scan an operator window (Figure 2) from left to right, and top to bottom, across the input image. Whenever we find a new solitary pixel at the center P of the operator window, we assign it a new object ID i, and make an entry $m[i] = i$ in the adjacency map. If we find multiple adjacent pixels in the window, we assign the current pixel P the object ID j of an already labeled pixel in the window.
   In parallel to this process, the adjacency list is being maintained and evaluated: If we find that P connects two areas formerly considered unconnected (e.g., P0 and P1 being '1' with P2 being '0'), we have to make a correction in the adjacency map. As outlined above, we create an entry in the adjacency list as follows: If P connects different object IDs l_{P1} and l_{P0} for pixels P0 and P1, we enter the label pair (l_{P1}, l_{P0}). If P connects different object IDs in P3 and P1, we enter the label pair (l_{P1}, l_{P3}). As soon as the main labeling process (see last paragraph) relinquishes access to the adjacency map, entries (a,b) in the adjacency list are used to correct the map m such that $m[a] = b$.
   With the sample input image in Figure 1, this procedure yields the pixel labeling shown in Figure 3, and the adjacency map in Figure 4. This is the final state after the two pairs $(3,2)$ and $(2,1)$ from the adjacency list have been applied.

Figure 3. Pixel labeled input image

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 2 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 2 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 2 0 0 3 3 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 1 1 1 0 2 2 2 2 2 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 4 4 4 4 4 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Object ID	Adjacent to
1	1
2	1
3	2
4	4

Figure 4. Adjacency map after pixel labeling

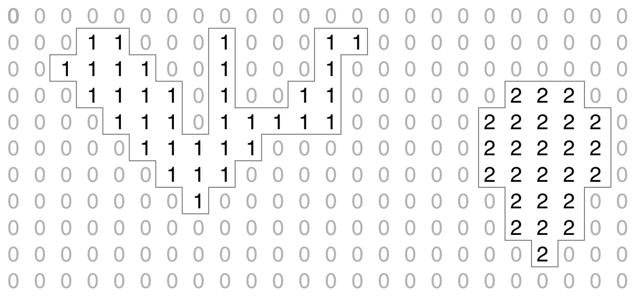
2. **Transitive flattening**. In this phase, we flatten each entry in the adjacency map into its smallest transitively reachable object ID (Figure 4). For the example, this creates the flattened adjacency map of Figure 5. Furthermore, in this new implementation, we also renumber all object IDs to be consecutive.

Object ID	Adjacent to	After renumbering
1	1	1
2	1	1
3	1	1
4	4	2

Figure 5. Flattened and renumbered adjacency map

3. **Object merging**. Finally, we use the flattened adjacency map to merge object segments that were transitively adjacent into single objects, assigning them the renumbered object ID found during flattening (Figure 6).

Figure 6. Merged object segments using flattened adjacency map

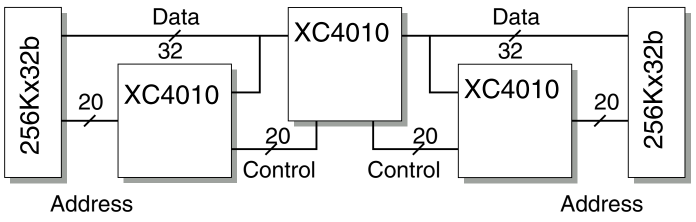


As result of these steps, the image has been labeled: Each of the 16b words now contains the object number this pixel belongs to. Furthermore, the number of labeled objects can be determined by looking at the last entry in the adjacency map (highest object ID, in the example equal to 2).

3 Platform Architectures

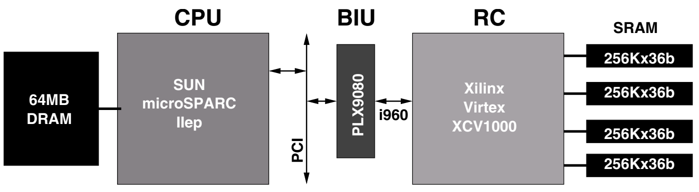
The initial implementation presented in [1] was intended for execution of the SPARXIL co-processor [5]. SPARXIL contained three at that time state-of-the-art Xilinx XC4010 FPGAs. Two of the FPGAs acted as address generators for access to two 256Kx32b static memory banks, while a third one was able to accept data streams from both of

Figure 7. SPARXIL co-processor architecture



these memories. Initially (1993), the specification called for FPGAs with a logic-block propagation delay of 4.5ns (speed grade -5). For comparison purposes, we will also give performance data for a realization using later FPGAs of the same type, but with a propagation delay of only 1.3ns (speed grade -1).

Figure 8. ACE-V co-processor architecture



The configurable computer ACE-V [6] combines a reconfigurable compute unit (RCU, realized by a Xilinx Virtex 1000 FPGA) with a conventional RISC processor (SUN microSPARC-IIep). The RCU has access to four independent ZBT SSRAM 256Kx36b memory banks and via a bus interface unit (BIU) to 64 MB of DRAM shared with the CPU. For purposes of this application, only the static memories are used. The FPGA used has a propagation delay of 0.8ns per logic block (speed grade -4). A dedicated RTOS executes on the microSPARC and provides system services such as file I/O.

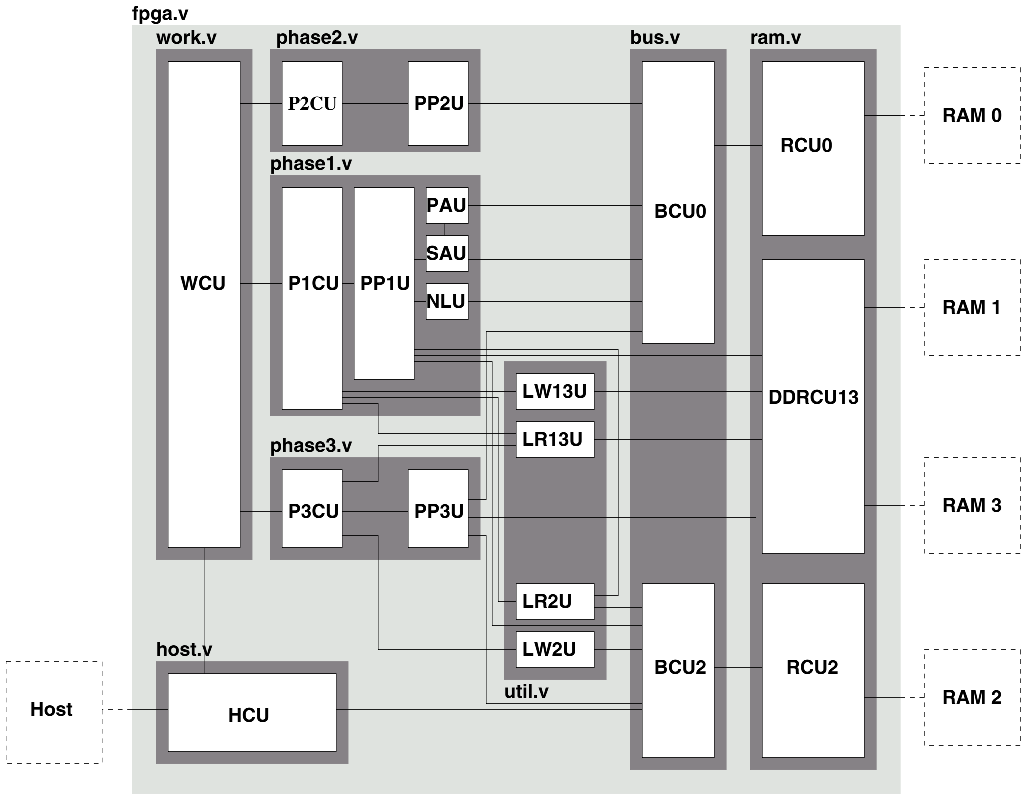
The well-known Splash-2 architecture is described in detail in [7]. For our purposes, it consists of 17 XC4010 FPGAs that are connected both in a fixed manner as a systolic array and additionally in a variable manner using a configurable crossbar interconnect. Each of the FPGAs has a local memory of 256Kx16b.

4 Design Flow

For implementing this algorithm in hardware, we used a HDL-based design flow that synthesized a Verilog description of the hardware components using Synplicity's Synplify tool. The resulting net list was then automatically processed by the Xilinx ISE

5.1 suite. The software parts of the application (file I/O, etc.) were formulated in C and compiled for the microSPARC-IIep CPU using the GCC compiler.

Figure 9. Hardware architecture



5 Hardware Implementation

Our revised hardware is implemented according to the architecture shown in Figure 9. The Work Control Unit WCU controls the global execution of the algorithm. It accepts a start command from the CPU and then hands control to the local controllers P1CU, P2CU and P3CU for each processing phase. The WCU also supervises the Host Control Unit HCU, which in turn accepts slave-mode data transfers initiated by the CPU from and to local memories (this is suspended during RCU execution). The HCU also accepts parameters from the host such as the image dimensions and whether to renumber labels. Furthermore, it also makes per-phase profiling data available to the host for benchmarking (see Section 6).

Each of the three processing phases relies on a highly optimized pipeline unit (PP1U, PP2U and PP3U) to actually perform the computations. In Phase 1, the background task for managing and evaluating the adjacency list is implemented in parallel to the pipeline: The NLU provides the next label to use and enters it in the adjacency map, the SAU stores a adjacency correction in the adjacency list, and the PAU processes list entries for correcting the adjacency map.

Phase 1 and 3 (which actually operate on the image) rely on the dedicated address generators LW13U, LR13U, LR2U and LW2U for storing and retrieving image data. These units also handle special cases such as clipping on the edges of the image. The memory accesses themselves are processed in two bus control units BCU0 and BCU2 (multiplexing address and data lines) and their associated RAM control units RCU0 and RCU2 (bidirectional to unidirectional bus conversion, delay write data according to ZBT SSRAM access protocol). DDRCU13, the RAM control unit responsible for accesses to RAM banks 1 and 3, is a special case: It uses both banks to present an external view of a single bank capable of *simultaneously* reading and writing to alternating even/odd addresses. This access pattern is always used when scanning over the image in Phase 1 and 3.

Figure 10. Data flow and memory organization

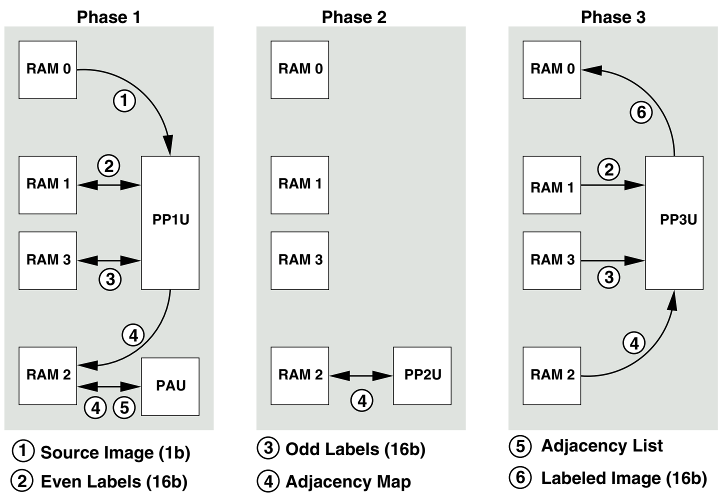


Figure 10 shows how these units interact in the different phases to maximally exploit the four independent memory banks on the ACE-V.

6 Evaluation

Platform	RCU	RAMs	Area[Cells]	Clock[MHz]	Time[ms]
SPARXIL-5	3x XC4010	2	1306	10.9	74
SPARXIL-1	3x XC4010	2	1306	25.4	32
Splash-2	9x XC4010	9	???	10.0	33
ACE-V	1x XCV1000	4	2101	36.0	15

Table 1. Area and performance comparison

Table 1 compares the various realizations examined. As discussed in Section 3, the two SPARXIL versions differ in their CLB propagation delay. Where available, we present some more detailed area requirements by listing the number of cells (4-LUT and associated flip-flop) used. The execution is that required for processing a 512x512 image.

The Splash-2 [3] version is not fully comparable with the SPARXIL and ACE-V solutions, since it is optimized for real-time processing of a stream of video frames at 30 fps. To this end, it uses duplicated hardware for Phase 2 and 3 of the algorithm to process two video frames in parallel. Only in this manner does the solution achieve the 33ms per frame processing time shown in the table.

Figure 11. Performance comparison with RISC and CISC CPUs

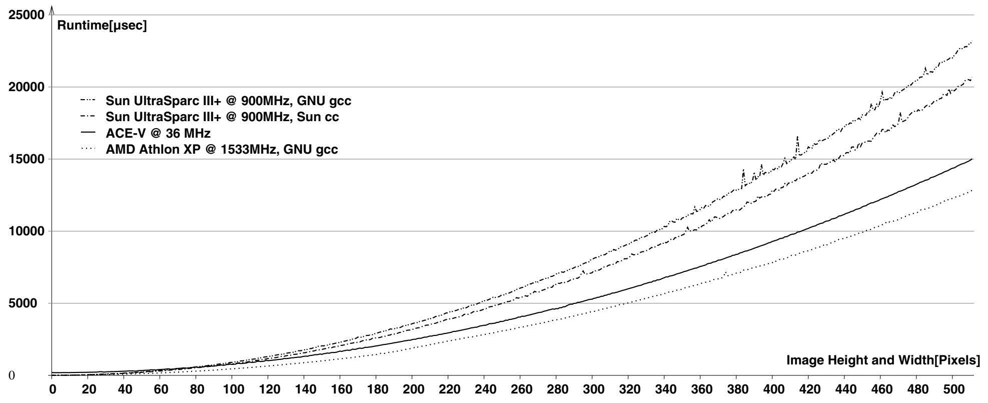
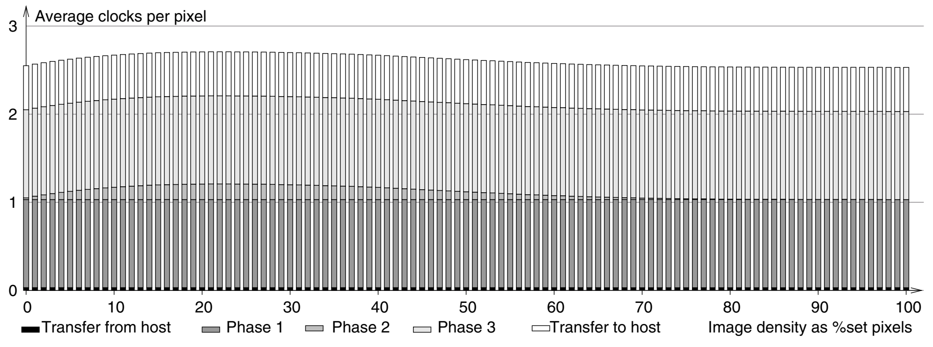


Figure 11 shows the execution time in relation to the image size processed. This compares current RISC (SUN UltraSPARC+ 900MHz) and CISC CPUs (AMD Athlon XP 1533MHz) to the ACE-V reconfigurable computer running at 36 MHz. The ACE-V easily beats the RISC even when a highly optimizing compiler (SUN cc with -fast option) is being used. The AMD CISC running at 42.6x the clock frequency of the RCU is actually faster, but only by a factor of 1.16.

From a more practical perspective, our current implementation would fit in a Xilinx XC2S300E commodity FPGA, a part currently priced at USD 53 in single quantities and which does not require any active cooling during operation. The design is I/O limited, the actual logic would fit a XC2S100 device.

Figure 12. Data-dependency of execution time



In Figure 12, we examine the computation time (as average clocks/pixel) in relation to the nature of the input data (image density randomly increasing from completely blank to completely filled). As can be seen, the transfer of the image to RCU-local memory is data-independent (1/32 clock/pixel). The same applies to Phase 1 which always takes 1 clock/pixel. Only Phase 2 is data-dependent, requiring an amount of time increasing with the number of separate objects in the image. After topping out at roughly 25% density, the number of separate objects (and thus the compute time for Phase 2) drops again. Phase 3 always takes 1 clock per pixel and the transfer of the labeled image back to the host memory requires 1/2 clock per 16b label (due to the 32b bus).

7 Future Work

The performance of the ACE-V based solution can be improved even further: The factor limiting the clock speed is currently the speed of the HCU and its connection to the *external* BIU. After the redesign presented here, the *internal* computation pipelines PP1U, PP2U and PP3U have much shorter critical paths than in [1]. Thus, the entire design would be amenable to running these parts of the circuit as well as the external memories at a doubled clock in the 50-60 MHz range. This would considerably speed up Phases 1 to 3 of the algorithm (see Figure 12).

Furthermore, note that the RCU currently uses the slowest speed grade -4 of the Virtex 1000 FPGA (0.8ns propagation delay through a 4-LUT). Switching to a more current device such as a Virtex II in speed grade -6 would allow for further gains (0.35ns propagation delay). Assuming a linear scaling of performance following this decrease and disregarding the limitations of the current ACE-V BIU (32b@33MHz PCI), this would reduce the processing time for a 512x512 image down to 6.5ms, beating even the fastest CISC processors currently available.

8 Conclusion

This work presented a refinement of our work on using a reconfigurable computer for accelerating the application of labeling connected regions in black and white images. The performance of the RCU-based solution easily surpasses a state of the art RISC processor and is extremely competitive even with regard to a modern CISC processor. The current hardware implementation has considerable head room for further performance improvements both on the architecture (doubled clock) as well as device (faster speed grade) levels.

References

1. Koch, A., Golze, U., “Practical Experiences with the SPARXIL Co-Processor”, *Proc. 31st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove (CA), 1997

2. Meyer, K., “Entwurf eines FPGA-basierten Co-Prozessors zur Objekt-Etikettierung in der Bilderkennung”, *diploma thesis*, Tech. Univ. Braunschweig (E.I.S.), Germany, 1997

3. Rachakonda, R.V., Athanas, P.M., Abbott, A.L, “High-Speed Region Detection and Labeling using an FPGA-based Custom Computing Platform”, *Proc. Field Programmable Logic and Applications (FPL)*, Springer, 1995

4. Wahl, F.M., “Digitale Bildverarbeitung”, Springer, 1989

5. Koch, A., “A Universal Co-Processor for Workstations”, in *More FPGAs, eds. Moore, W., Luk, W.*, Oxford 1994

6. Koch, A., Golze, U., “A Comprehensive Prototyping Platform for Hardware-Software Codesign”, *Proc. Workshop on Rapid Systems Prototyping*, Paris, 2000

7. Arnold, J.M, Buell, D.A., Davis, E.G., “Splash 2,” *Proc. Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992