RANDOM DECISION TREE BODY PART RECOGNITION USING FPGAS

Jason Oberg

Computer Science and Engineering
University of California, San Diego
San Diego, CA USA
email: jkoberg@cs.ucsd.edu

Ken Eguro, Ray Bittner, Alessandro Forin

Embedded and Reconfigurable Computing
Microsoft Research
Redmond, WA USA
email:{eguro, raybit, sandrof}@microsoft.com

ABSTRACT

Random decision tree classification is used in a variety of applications, from speech recognition to Web search engines. Decision trees are used in the Microsoft Kinect vision pipeline to recognize human body parts and gestures for a more natural computer-user interface. Tree-based classification can be taxing, both in terms of computational load and memory bandwidth. This makes highly-optimized hardware implementations attractive, particularly given the strict power and form factor limitations of embedded or mobile platforms. In this paper we present a complete architecture that interfaces the Kinect depth-image sensor to an FPGA-based implementation of the Forest Fire pixel classification algorithm. Key performance parameters, algorithmic improvements and design trade-off are discussed.

1. INTRODUCTION

A hallmark of the Microsoft Kinect system is its high-quality vision-based object recognition. An important and computationally-intensive part of this processing is pixel classification using random decision trees [6, 7]. In the existing implementation, the camera is simply a sensor and classification is performed entirely in software. On a standard PC, classification can fully occupy a modern CPU in order to maintain the desired 30 frames per second.

As the use of vision-based systems becomes more ubiquitous (e.g. as a generic natural user interface with more sophisticated processing, etc.), software-based classification may not be sufficient. This is certainly a problem for low-end embedded platforms.

Specifically, we need to consider two aspects of the random decision tree processing:

Efficiency – Constant active use means that the computation must be efficient. This is both in terms of power consumption and in terms of the use of shared resources, particularly bandwidth to external memory.

Performance – Fast computation with guaranteed worst-case performance can improve interactive experiences and allows more time for other processing.

These considerations make it attractive to implement decision tree processing directly in hardware. Towards this end, FPGAs are an ideal computational platform. The flexibility of FPGAs gives us two advantages. First, we can update the classification algorithm itself to accommodate new innovations or to deal with different use cases. This flexibility is particularly important since we are just beginning to understand the real-world demands on, and applications of, a practical consumer-grade vision system.

Second, modern FPGAs are large enough that they can encapsulate full systems. The remaining portions of the vision pipeline and the ultimate consumer of the data (end-user applications) can be placed alongside the decision tree processing and gain the power/performance advantages of direct hardware implementation.

The main contributions of this paper are:

- The first Verilog implementation of the Kinect decision tree pixel classification algorithm (Forest Fire).
- Novel algorithmic and architectural optimizations for processing randomized decision trees that account for the limitations of real-world memory systems.
- A framework for developing and debugging vision algorithms on the Xilinx ML605 board. This system can interface with the Microsoft Kinect SDK using live or stored Kinect camera images.

2. FOREST FIRE BACKGROUND

The Forest Fire algorithm uses random decision trees and forests [4, 8] to classify pixels from the Kinect depth camera into parts of the human body.

Shown in Fig. 1, every pixel in the input frame traverses several binary trees. Starting at the root of a given tree, a decision is made to proceed to either the left or right child, based on an evaluation function:

(a) does a target pixel in the image surrounding the current pixel belong to the same player object? (the relative position of this target pixel is defined by the tree node the current pixel is visiting)

(b) if so, is the distance in depth between these two pixels above a threshold? (again, defined by the current tree node)

(c) if so, move to the right child, else to the left child.

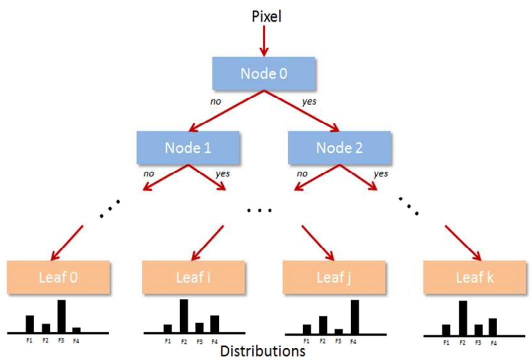

Fig. 1. Decision tree/forest structure

```
double[] ClassifyDT(node, v)
   if node.IsSplitNode then
      if node.f(v) >= node.t then
         return ClassifyDT(node.right, v)
      else
         return ClassifyDT(node.left, v)
      end
   else
      return node.P
   end
end
```

Fig. 2. Forest Fire algorithm

Eventually, this traversal will reach a leaf node. A leaf node contains the probability that any pixel reaching that leaf belongs to a particular human feature, such as a head or foot. The pixels are classified by every tree and the probabilities for each pixel are aggregated across all trees in the forest.

More formally, decision tree processing is summarized in Fig. 2. Note that the decision procedure *f*, the number of trees, and the database structure and values (interior nodes and leaf probabilities *P*) are determined *a priori* and are application dependent. Determining these factors is part of the training process for the decision tree and is outside the scope of this paper. For more information the reader is referred to [4].

Because the structure of the system is use-dependent, the flexibility of an FPGA implementation is an important advantage over an ASIC. Applications in vision, speech, or web search engines would all require different databases and procedures. Even if we were to take one application, in this case computer vision, and retrain the database for detection of a different object, the decision procedure and database structure may be different, even if the basic algorithm operates in a similar manner.

3. RELATED WORK

To the best of our knowledge, hardware systems for classification using randomized decision trees have received minimal attention. Most related work from the computer vision community is specific to software and does not consider the issues and advantages of implementing it in hardware.

One work by Narayanan et al. [1] uses FPGAs to accelerate the construction of decision trees. Their hardware implementation focuses on calculating the *Gini Score* – a quantity used to calculate thresholds in the decision tree. As mentioned earlier, this is part of the training process. While a required complement to using the decision trees for classifying data, it is not the focus of this paper.

Recent work by Becker et al. [2] uses decision trees to accelerate object tracking on FPGAs. This work is the most similar to ours, but there are significant differences in the algorithms used in the two works. Specifically, the decision computation used in [2] is more complex than ours, while the trees are much simpler. Thus, [2] focuses heavily on parallelizing the classifier decision computation (and converting the input data into a more efficient internal representation), while this paper focuses on the efficiency of data movement and the traversing the trees themselves.

Other high-performance computing devices, such as graphics processing units (GPUs), have also been used to accelerate the evaluation of decision trees. Specifically, a paper by Sharp [3] describes how to parallelize the evaluation of decision trees using a GPU to achieve a ~100x speedup over conventional CPU versions. However, our main goal is to perform the computation directly in hardware to produce a fast, yet low-power embedded system. This type of direct hardware implementation is necessary for platforms where a power-hungry GPU with hundreds of stream processors would not be practical.

Decision trees have also been implemented in FPGAs for other applications such as Internet Protocol (IP) lookup. For example, Hoang Le et al. [5] presents an architecture for quick packet routing. They heavily pipeline the packet traversal through the binary tree and take advantage of dual ported Block RAM (BRAM) and caching to optimize performance. Our architecture is quite different, since our database is much larger and, thus, greatly exceeds the capacity of on-chip BRAM and must reside in external DDR. This fundamental architectural difference means that our work focuses heavily on minimizing and maintaining sequential access to the DDR to optimize throughput. As will be discussed later, we accomplish this by introducing a new computation and storage element – a continuously sorted FIFO.

4. FOREST FIRE MEMORY OPTIMIZATIONS

As described in Section 2, the number of steps required to process a frame is the product $N*T*L$, where N is the number of pixels to be classified, T is the number of trees, and L is the number levels in each tree (assuming all trees are of identical depth). In the Kinect system, N is between 0

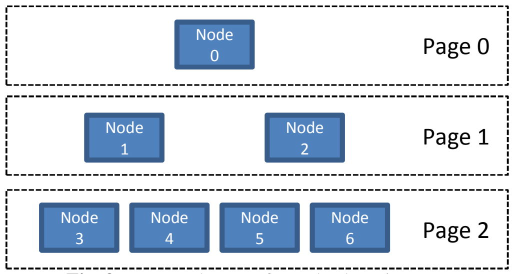

Fig. 3. Memory layout of tree database in DDR

and 19,200 (as will be discussed in Section 5, the 160x120 input image is filtered prior to classification), *T* equals 3, and *L* equals 20. Among these steps, the traversal of each pixel through any given tree is independent. Thus, the algorithm can theoretically be parallelized *N***T* ways.

However, looking at the evaluation function described in Section 2, there are two critical memory accesses that need to be made at each step. These memory accesses will bottleneck a realistic physical implementation in which the memories have finite bandwidth. The first memory access is to the current tree node in the database. This database lookup retrieves two values: a pixel offset and a comparison threshold. The second memory access is to the original depth image at an address based on the pixel offset obtained from the database lookup.

The input depth image is relatively small (~38KB) and thus can be held in BRAM within the FPGA. However, the database for the Forest Fire algorithm is comparatively large (25MB) and thus must be held in an external memory, such as DDR. As these are random decision trees, there is little to no *a priori* discernible correlation or pattern in either the offset or threshold values in the database. Because of this, the prescribed tree node must be directly consulted for every decision.

Since bandwidth to external DDR is a limited resource, the achievable performance of the algorithm is heavily bounded by the number of lookups made to the database. Furthermore, in the case of a System-On-Chip, the pixel classification will be merely one component that must share access to the DDR with other computations. For these reasons, we took a close look at ways to minimize the number of external database memory accesses.

4.1. Traversal Order

While the algorithm specifies that all trees in the forest must be traversed to collect the probability distribution, as mentioned earlier, each pixel proceeds independently of all other pixels for all trees. Consequently, the order of traversal can be changed without affecting the results.

Depth-first traversal, where a single pixel is processed completely from root to leaf for a given tree before the next pixel is processed, can be attractive for light-weight implementations. This is because very little state needs to be stored (i.e. if we have one in-flight pixel, we only need a single pointer to its current tree node). However, this algorithmic decision has an impact on the number of external memory accesses that are required, along with the amount of memory bandwidth consumed.

For example, DDR modules are organized into "pages". Consecutive accesses within a page are much faster and, thus, consume less bandwidth than accesses that cross a page. In the case of the DDR3 DIMM on the Xilinx ML605 board (Micron MT4JSF12864HZ-1G4D1) mated with Xilinx's Memory Interface Generator version 3.61 controller used in our experiments, we observe that fully pipelined in-page reads can be serviced at a rate of one read every other cycle. Conversely, each read request that crosses an 8KB page boundary creates a 10 cycle bubble in the pipeline to account for the latency associated with opening a new page.

Since any given tree is large and spans many pages, depth-first traversal guarantees page misses during the processing of every pixel. Consider the simple example in Fig. 3. Here, each level of the database is held on a unique page (the actual database used is densely packed – this arrangement is used merely for illustrative purposes). In this case, each depth-first traversal will send three read requests, forcing three page changes: the *level 0* node, the *level 1* node and the *level 2* node. Processing all pixels completely will require *N***T***L* read requests which will generate *N***T***L* page changes.

As an alternative, we can traverse the tree in a breadth-first manner. In this case, anywhere between two and *N* pixels can be batched together, processing all pixels in the batch at a single tree level before proceeding to the next tree level. The rationale behind this change is that, regardless as to their specific traversal paths, pixels that are at the same level in the tree will need to access database nodes with addresses that are near one another (this is more true in earlier levels, where the tree is narrow, and less true in later levels, where the tree is very wide). Thus, traversing breadth-first can amortize read requests and page changes among the computations for multiple pixels.

If we assume maximum breadth-first traversal, batching all *N* pixels together, the example in Fig. 3 will create at most one read request and one page miss at *level 0* (all pixels start at the root, so after we read *node 0* once, we do not need to read it again), at most *N* read requests and one page miss at *level 1* (in the worst case consecutive pixels alternately traverse to *node 1* and *node 2*, requiring a unique read request, but both nodes are on the same page) and at most *N* read requests and one page miss at *level 2* (similarly, consecutive pixels may traverse to different nodes, but all nodes at this level are on the same page). To process all pixels in a maximal breadth-first manner will require at most *N**((*T***L*) – 1) read requests (and likely far fewer) but only *T***L***A* page changes, where *A* is the average number of pages required to hold the nodes on a given level. *A* will be less than one towards the top of the tree and greater than one towards the bottom of a large tree.

Although breadth-first traversal reduces the number of memory transactions considerably, this approach also requires more intermediate state (i.e. we must keep separate current tree node pointers for each pixel in the batch). Keeping the full 19,200 pointers needed for maximal breadth-first traversal of the entire image requires ~57KB. As we will show in Section 6, this is well within the capabilities of onboard BRAM for our hardware implementation. Thus, this hardware investment is worthwhile considering the bandwidth savings to the external DDR. Furthermore, as we will discuss in Section 7, the batch size can be reduced without affecting nominal performance to lower the onboard memory requirements.

That said, storing this table can have performance implications for a software implementation running on a traditional CPU. Unlike the hardware-based computation where we can have highly customized memories, this approach does not improve performance for our software implementation. This is likely because the additional memory traffic created by the pointer table accesses offsets savings in database lookups.

4.2. Pixel Sorting

As shown in our discussion in the previous section, although breadth-first traversal can reduce the number of read requests and page changes, the same address can be requested multiple times, resulting in unnecessary memory transactions (e.g. alternating between *node 1* and *node 2*). By extension, this means that the same page can be opened multiple times when the nodes for a tree level span more than one page. In the database used in our system, this begins in the trees at level 10 (of 20 total).

Ideally, each time we request a given database node, we would like to process all of the pixels that require that node consecutively, reading the node once, using it to process as many pixels as we can, and never requesting it again until the next complete iteration through the tree. In our example from Fig. 3, this would mean somehow bundling all of the pixels that traverse to *node 1* into one group and all the pixels that traverse to *node 2* into another, before we begin processing the pixels through the tree at *level 1*. Similarly, each time that we open a new page, we would ideally like to process all of the pixels that need database nodes from that page consecutively. In this way, each unique page would get opened once, as many same-page accesses would be made as possible, and when we move to a new page, the previous page would not get opened again until the next iteration through the tree.

This can be accomplished by traversing the tree in a breadth-first manner while also keeping the pixels sorted as they are processed – ordered by their respective current tree node from smallest to largest. With this sorting, each time we attempt to process a pixel we know that it will either require the same database node as the previous pixel or a database node at a higher address. Similarly, if this pixel requires a database node on a new page, we are guaranteed that all future accesses during this iteration will be to database nodes on the same page (at a higher address) or a different page (also with a higher address).

Fig. 4 shows a simple example with four pixels of how this sorting can be accomplished. Note that for clarity, we depict the list of pixel tree node pointers as a linear FIFO that is much larger than the nominal size. In our actual implementation, a circular FIFO of size *N* is used. The process for *level 0* begins in Fig. 4a with four pixels at the root, *node 0*.

Fig. 4 also shows three pointers, indicating the *head* of the list (denoting the next pixel to be processed in the current level), the *left* (denoting the space just beyond the last computed pixel that traversed to its left child), and the *right* (denoting the space just beyond the last computed pixel that traversed to its right child). As processing has not yet begun in Fig. 4a, the *left* and *right* both point to the first available space in the FIFO.

In Fig. 4b, we show that the first pixel traverses the tree right to *node 2*. This pixel is pushed to the *right* and the pointer is incremented. Shown in Fig. 4c, the next pixel also traverses the tree to its right child. Shown in Fig. 4d, the third pixel traverses to its left child, *node 1*. To maintain proper sorting, the existing pixel at the *left* pointer is evicted to the *right* and the new pixel is inserted at the *left*. As both the *left* and *right* were written, both pointers are incremented. This is shown in Fig. 4e.

A single read-modify-write insertion sort operation is sufficient to always keep the pixels fully ordered, but this requires one additional consideration when managing the *left* pointer – each time the *head* pointer encounters a new tree node (when the current *head* node is greater than the

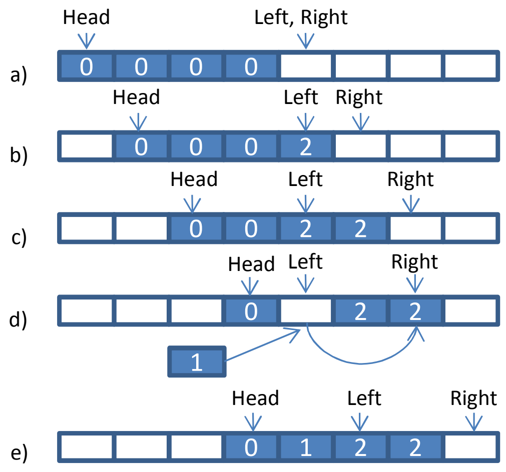

Fig. 4. Basic sorting

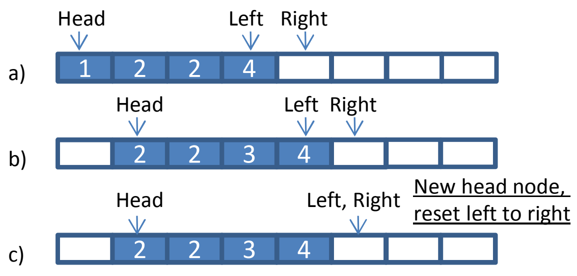

Fig. 5. Management of *left* pointer

Table 1. Memory accesses for various tree traversals.

Traversal	*Cached Hits*	*Page Hits*	*Page Misses*	*Norm. BW*
DF Avg	25,551	79,641	128,568	1.000
BF Avg	159,838	50,523	23,399	0.232
BFS Avg	218,536	13,709	1,515	**0.029**
DF Stress	124,314	394,086	633,600	1.000
BF Stress	892,202	173,111	86,687	0.170
BFS Stress	1,119,005	30,714	2,281	**0.012**

previous *head* node), we must reset *left* to *right*. Consider the situation in Fig. 5. In Fig. 5a, we have begun processing pixels at *node 1*, producing children at *node 3* or *node 4*. In Fig. 5b, we discover we have processed all of the pixels that were at *node 1* and we have begun processing those at *node 2*. Since the children of pixels at *node 2* will be either *node 5* or *node 6*, any children from this point forward must enter the FIFO behind all existing pixels. Thus, as shown in Fig. 5c, the *left* pointer must be reset to the *right* pointer.

This sorted breadth-first traversal only requires one additional pointer (*left*) on top of the intermediate storage that is needed to perform unsorted breadth-first traversal. However, it also requires additional computations and in-line read-modify-write operations to the list of tree node pointers to keep the database accesses sorted. As we will see in Section 5, this can be easily done without a performance penalty in the direct hardware implementation because the FPGA can exploit fine-grain parallelism and multi-ported or double-clocked memories. However, as with the unsorted breadth-first traversal, this approach does not improve the performance of a traditional software implementation. This is likely due to the additional computations and memory activity caused by sorting.

Returning to the example in Fig. 3, sorting reduces the number of read requests and page misses to the absolute minimum. That said, as with the unsorted breadth-first traversal, the precise number of memory accesses and page misses is wholly dependent on the input data.

4.3. Memory Bandwidth Evaluation

Since the behavior of the breadth-first and sorted breadth-first optimizations is data dependent, before building any hardware we wanted to quantify the potential memory bandwidth advantages of these techniques for real data. As shown in Table 1, we performed a detailed analysis of the database accesses made when processing two test images using the three traversal approaches.

The first image, *Avg*, represents an image with an average number of pixels to be classified. This image shows two average-sized adults head-to-toe at the suggested camera-user distance. The second image, *Stress*, represents an image where all pixels in the frame must be classified. Although this is a degenerate case that will only be created when an object strays very close to the camera, it represents the maximum computational load. The *Avg* image accesses 233,760 tree nodes (3896 pixels * 3 trees * 20 levels) and the *Stress* image accesses 1,152,000 tree nodes (19,200 pixels * 3 trees * 20 levels).

In Table 1, *DF* indicates depth-first traversal, *BF* indicates maximally batched breadth-first traversal and *BFS* indicates maximally batched breadth-first traversal with sorting. This table also shows four metrics. *Cached Hits* indicates the number of database accesses that that did not require an external memory access. This is either because the node requested was the same as the previous node or because the node requested was fetched along with the previous memory access. While each database node in our system is 64 bits, every access to the DDR3 in our test platform returns a 512-bit word. This 512-bit word is used as a simple single-entry cache in our hardware implementation.

Page Hits indicates the number of database accesses that required an external memory access but requested a value within the same page as the previous access. *Page Misses* indicates the number of external memory accesses that resulted in a page crossing. *Norm. BW* indicates the amount of memory bandwidth consumed (normalized to the depth-first results), considering that a *Cached Hit* does not require an external memory access, a *Page Hit* occupies the memory for two cycles when the controller has a full pipeline of outstanding read requests, and a *Page Miss* occupies the memory for ten cycles when the controller has a full request pipeline.

Looking at Table 1, we see that breadth-first traversal reduces the consumption of memory bandwidth considerably. The bandwidth needed for breadth-first traversal is only ~20% that of the depth-first traversal. Breadth-first traversal with sorting further reduces memory bandwidth consumption to only 1-3% as compared to depth-first traversal. This is roughly an additional order of magnitude improvement beyond the unsorted breadth-first results.

As mentioned before, the lessons learned from this evaluation looking towards a hardware implementation cannot be directly translated for a software version. Although memory bandwidth is also a critical resource in traditional processors, traversal order and sorting causes

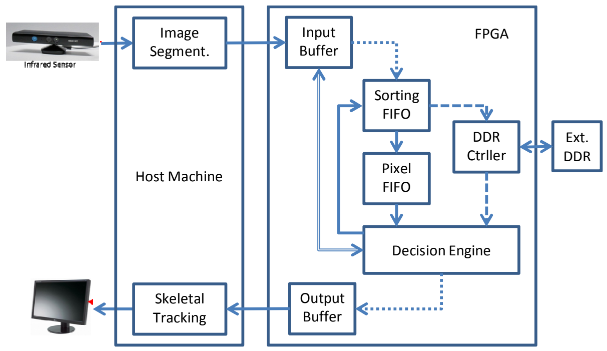

Fig. 6. Kinect processing pipeline and Forest Fire hardware

other effects due to differences in parallelism, caching, etc. The most highly optimized software implementation we use for comparison in Section 6 uses depth-first traversal. Depth-first traversal is also used in the GPU implementation in [3].

5. FOREST FIRE HARDWARE ARCHITECTURE

Since there are clear bandwidth advantages to sorted breadth-first tree traversal for hardware-based systems, we used this approach for our FPGA-based implementation of the Forest Fire algorithm. Fig. 6 shows a block diagram of how this hardware fits into the full Kinect vision pipeline, interfacing the FPGA with a Kinect camera and other parts of the existing Kinect SDK.

The system begins with a Kinect sensor connected via USB to a host PC. In addition to handling the USB 2.0 protocol to receive the raw depth image from the camera (this cannot be handled sufficiently using the onboard USB 1.0 controller present on the ML605 board), the host machine also performs the first stage of the Kinect vision pipeline: *image segmentation*. This process identifies the pixels representing moving objects, separating them from the static background. The depth image with these tagged "active" pixels is sent to an input buffer on the FPGA.

Once the input buffer has been filled, the FPGA performs the Forest Fire algorithm. This phase classifies the active pixels in the depth image, producing the probabilities for thirty-one distinct body parts for each pixel. These probabilities are written into an output buffer on the FPGA which are sent back to the host PC.

At this point, the remaining part of the Kinect SDK resumes operation to perform skeletal tracking. This process connects the body parts identified by the classification phase into a human skeleton either for display or to be passed on to a user application (e.g. a game or other target for the natural user-interface).

5.1. FIFO Initialization and Classification

When the image is first transferred to the FPGA, the hardware must "seed" the sorting FIFO with pixels to begin the tree-based classification process. In our current implementation, the image is fully read from the input buffer in scan-line order. The pixels are filtered based on the "active" designation produced by the image segmentation phase and only active pixels enter the sorting FIFO for processing.

Once the FIFO is loaded, the hardware begins executing the classification algorithm on the pixels in the FIFO. The compute logic pops a pixel from the sorting FIFO, looks up the node it wishes to access from the database stored in DDR, computes the appropriate target pixel address in the original image (still held in the input buffer) based on the offset from the database node, retrieves the comparison pixel, computes the evaluation function and decides whether the pixel should be pushed back into the FIFO now pointing to the left or right child. These pixels are sorted upon push-back as described in Section 4.2.

Our implementation is heavily pipelined. The system utilizes multiple FIFOs (i.e. to/from the DDR controller and sorting FIFO) to mitigate the latency of the DDR. Although, as discussed earlier, the controller can accept new read requests every other or every tenth clock cycle, the overall latency through the controller and DDR can be up to roughly 100 clock cycles. These FIFOs allow the system to issue database reads far ahead of the computations performed in the decision engine.

As discussed in Section 4.2, if the pixel moves to the right child, it is pushed to the FIFO at the *right* pointer. However, if the pixel moves left, the pixel must be inserted at the *left* pointer, displacing the current entry that must be written to the *right*. The possibility of one read/one write to the *left* followed by a write to the *right* is handled in the current implementation by clocking the sorting FIFO at twice the speed of the rest of the computational pipeline.

When a pixel reaches a leaf node, the resulting probabilities are sent to the output buffer and the pixel is done with the current tree. After all of the pixels have traversed the first tree, the pixels are left in the FIFO. The FIFO does not need to be reinitialized to begin processing the next tree, since all pixels will begin at the root of the subsequent tree. Thus, we can simply restart the processing, performing an in-line replacement of the tree node pointers with the address of the appropriate root node. Once the last tree is processed, the algorithm terminates, signaling the software that processing for the frame is complete.

6. EVALUATION

As discussed in Section 1, the primary goal of this work was to build an efficient direct hardware implementation of the Forest Fire algorithm. We prototyped our system using the Xilinx ML605 board containing a Virtex-6 LX 240T.

Table 2. Resource Utilization (V6 LX240T).

	LUTs	*FF*	*BRAM*
Full System	11,319 (7.5%)	13,327 (4.4%)	125 (30.0%)
· Forest Fire Core	2,511 (1.7%)	2,267 (0.8%)	35 (8.4%)
Sorting FIFO	*457 (0.3%)*	*77 (0.0%)*	*33 (7.9%)*
· DDR3 Controller	6,522 (4.3%)	9,935 (3.3%)	0 (0%)
· PC Interface	2,282 (1.5%)	1,123 (0.4%)	90 (21.6%)
Input Buffer	*248 (0.1%)*	*46 (0.0%)*	*10 (2.4%)*
Output Buffer	*151 (0.1%)*	*8 (0.0%)*	*77 (18.5%)*

Table 3. Hardware Performance.

Algorithm	*Avg. Cycles @ 75Mhz*	*FPS*
BFS Avg	349,492 (1.18x faster)	214
BF Avg	414,557 (1.0)	181
BFS Stress	1,349,706 (1.27x faster)	56
BF Stress	1,714,525 (1.0)	44

6.1. Hardware Characterization

The logic and memory utilization of our FPGA implementation is shown in Table 2. The entire system, including the Forest Fire core, the DDR3 controller, and the Ethernet controller used to communicate with the host represents a fairly small portion of the target platform. The logic and flip-flop utilization of the full system is less than 8% of the LUTs and 5% of the FFs. The largest fraction of the resources consumed on the FPGA is BRAM (30%). Even so, the computational core itself only uses 8.4% of the BRAM, entirely devoted to the sorting FIFO. The remaining BRAM are primarily consumed in the input and output buffers to the PC. We will discuss how we can reduce the BRAM requirements in Section 7.

We operate the Forest Fire core at 75MHz (as mentioned earlier, the sorting FIFO is run at twice this clock rate, 150MHz). The DDR3 is clocked at its minimum allowable frequency of 150MHz. Although a higher clock rate would result in higher performance, as we will show, the throughput of the system is already very high. A core frequency half of the DDR clock rate is used because, at best, the DDR can accept new same-page requests every other clock cycle.

The execution time of the computational core is shown in Table 3. Measured directly in hardware, the cycle count begins when the input buffer is full and continues until the output buffer receives the last leaf node. We tested the system 10 times with the *Avg* and *Stress* images from Section 4 and averaged the results of all runs. Given that the DDR currently has no other traffic to service besides requests from our Forest Fire core, the cycle counts were quite consistent – varying by no more than +/- 7 cycles.

Although we already proved that sorted breadth-first traversal was advantageous from the standpoint of memory bandwidth, we also measured the effect on runtime. This was performed by disabling the sorting aspect of the FIFO. For example, forcing the FIFO to always push pixels exclusively to the right, regardless as to which child node is selected, maintains the original scan-line order, as in the unsorted breadth-first search.

Looking at Table 3, we see that traversing the trees with sorted accesses requires ~350K cycles at 75MHz for the *Avg* image. This achieves a throughput of roughly 214 frames-per-second (FPS). This provides 1.18x faster performance as compared to the unsorted breadth-first traversal (181 FPS). Behavior is similar for the *Stress* image, with sorted breadth-first traversal achieving 56 FPS, or 1.27x better performance as compared to unsorted breadth-first traversal (44 FPS).

It is difficult to determine how the performance of the system might change if incorporated into an SOC where it shared access to the DDR with other computational elements. That said, given the excellent throughput, even in the degenerate *Stress* case, the existing implementation has quite a bit of headroom since the Kinect sensor captures images at 30 FPS.

6.2. Software Comparison

The direct hardware implementation described here is necessary for many applications because of the computational complexity of pixel classification. Although the existing Kinect SDK functions adequately on a modern desktop or laptop processor, software is no longer sufficient when run on an embedded processor.

For example, when run on a single-core 1.6 GHz Intel Atom 230, our most heavily optimized software implementation struggled to process the *Avg* image at 14.3 FPS. When faced with the *Stress* image, the performance dropped below 2 FPS.

The performance on a modern ARM processor is similar. We repeated the experiment with a Qualcomm MSM8960, a dual-core 1.0 GHz Cortex-A15. Running single-threaded, the Cortex processed the *Avg* image at 16.3 FPS and the *Stress* image at 3.0 FPS.

To meet 30 FPS, we would need two Atom or ARM cores at 100% utilization just for pixel classification in the nominal case – discounting stress cases, any other parts of the vision pipeline, or user applications. For power reasons alone, this places significant limitations on the systems we can build only using software-based processing.

Along the same lines, we do not consider performing pixel classification on a GPU because embedded GPUs do not have the GP-GPU capabilities found in standard PC graphics cards, such as those used in [3].

7. FUTURE WORK

Looking ahead, there are two particular aspects of our current system that we would like to investigate in the future. First, porting the implementation to a Spartan-class device, and second, integrating more of the Kinect pipeline into hardware.

Porting our system to a Spartan or similar FPGA is important for both cost and power reasons. While the ML605 was an excellent platform for a proof-of-concept, Virtex-class devices are designed for logic capacity and performance rather than cost and low power.

There is only one significant hurdle that prevents the existing implementation from fitting into a very low-cost, low-power device such as the Spartan LX-16 or possibly even the LX-9: block memory. Virtually all of the block memory in the existing implementation is consumed by the input buffer, the sorting FIFO and the output buffer. While the input image will likely remain on-chip, the size of the sorting FIFO and output buffer can be reduced considerably without effecting performance.

For example, the sorting FIFO is sized such that we can maximally batch all 19,200 pixels in the depth image. As mentioned in Section 4, though, it is rare that the number of active (non-background) pixels will comprise more than ~1/4th of the image. Thus, we can optimize the sorting FIFO by sizing it to handle the typical number of active pixels and simply run multiple classification iterations when the number of active pixels exceeds the capacity of the sorting FIFO. Multiple iterations may slightly increase the necessary processing time as compared to a system that can accommodate a larger batch size, but the existing implementation is more than fast enough to accommodate the expected penalty for atypical frames.

Discussion of the output buffer brings us to the second area we would like to investigate further - integrating additional parts of the Kinect pipeline. Beyond simply providing a more sophisticated system, integrating some or all of Skeletal Tracking will allow us to shrink the output buffer considerably. This is because the output of classification consists of multiple probabilities for each active pixel. The first stage of Skeletal Tracking aggregates this data heavily. Rather than describing individual pixels, the result of the first stage of Skeletal Tracking is a small handful of aggregated body part positions. Since all of the computations beyond this first stage only look at the body part positions, this dramatically reduces the working set. In this case, the output buffer would be less that 1KB rather than nearly 300KB.

8. CONCLUSION

Random decision tree classification is a powerful, but computationally expensive machine learning algorithm. Supporting random decision trees raises significant computing challenges, particularly for high duty-cycle embedded applications.

In this paper we demonstrated that the Forest Fire algorithm can be efficiently implemented in hardware. More importantly, we built a framework in which researchers can build and explore future hardware-based computer vision applications using this unique and easily accessible device. In addition, we showed that there is plenty of available room for future expansion, both in terms of resources on the FPGA and bandwidth to external DDR. We have already outlined new parts of the existing computational pipeline that can be readily migrated to hardware and we hope to eventually encapsulate full systems, including end-user applications.

9. ACKNOWLEDGMENTS

We would like to thank Shaoshan Liu for his help performing benchmarking on the ARM Cortex-A15.

10. REFERENCES

[1] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno. *An FPGA Implementation of Decision Tree Classification*. In proceedings of the Design Automation and Test Europe Conference (DATE), April 2007. Pg. 1-6.

[2] T. Becker, Q. Liu, W. Luk, G. Nebehay, and R. Pflugfelder. *Hardware-accelerated Object Tracking*. Computer Vision on Low-Power Reconfigurable Architectures Workshop, Field Programmable Logic and Applications (FPL), 2011.

[3] T. Sharp. *Implementing Decision Trees and Forests on a GPU*. In the European Conference on Computer Vision (ECCV), 2008. Pg. 595-608.

[4] A. Criminsi, J. Shotton, and E. Konukoglu. Decision Forests for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. Microsoft Technical Report. MSR-TR-2011-114.

[5] H. Le, W. Jiang, and V. K. Prasanna. *A SRAM-based Architecture for Trie-based IP Lookup Using FPGA*. In proceedings of the conference on Field Programmable Custom Computing Machines (FCCM), 2008. Pg. 33-42.

[6] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. *Real-Time Human Pose Recognition in Parts from a Single Depth Image.* In proceedings of the Computer Vision and Pattern Recognition conference(CVPR), Colorado Springs CO, IEEE, June 2011.

[7] Y. Amit, and D. German. *Shape Quantization and Recognition with Randomized Trees.* In Neural Computation Vol. 9 No. 2, October 1997. Pg. 1545-1588.