

# Parallel Object Recognition on an FPGA-based Configurable Computing Platform \*

Yongwha Chung

System Engineering Section  
Electronics and Telecommunications Research Institute  
Taejon, Korea

Seonil Choi and Viktor K. Prasanna

Department of EE-Systems, EEB-200C  
University of Southern California  
Los Angeles, CA 90089-2562  
{yongwha + seonil + prasanna}@halcyon.usc.edu  
<http://maarc.usc.edu>

## Abstract

*Object recognition involves identifying known objects in a given scene. It plays a key role in image understanding. Geometric hashing has been proposed as a technique for model-based object recognition in occluded scenes. However, parallel techniques are needed to realize real-time vision systems employing geometric hashing.*

*In this paper, we develop a design technique for parallelizing geometric hashing on an FPGA-based platform. We first transform the hash table which contains symbolic data into a bit-level representation. By regularizing the data flow and exploiting bit-level parallelism in hardware, our design achieves high performance. Using our approach, given a scene consisting of 256 feature points, a probe can be performed in 1.65 milliseconds on an FPGA-based platform having 32 Xilinx 4062s. In earlier implementations, the same probe operation was performed in 240 milliseconds on a 32K-node CM2 and in 382 milliseconds on a 32-node CM5. Also, the same operation takes 40 milliseconds on a 32-node IBM SP-2. By parameterizing the application and the device characteristics, we derive an area-time efficient design based on these pa-*

*rameters. Furthermore, our approach can be applied to many geometric hashing methods and is portable to other FPGA devices.*

## 1 Introduction

Object recognition is a key step in computer vision. Most model-based recognition systems work by hypothesizing matches between scene features and model features, predicting new matches, and verifying or changing the hypotheses through a search process. *Geometric hashing* [12] has been proposed as an alternate approach for object recognition. However, parallel techniques are needed to use geometric hashing in real-time applications.

In geometric hashing, a set of models is specified using their feature points [12]. For each model, all possible pairs of feature points are designated as a *basis set*. The coordinates of the features points of each model are computed relative to each of its basis. These coordinates are then used to hash into a hash table. The entries in the hash table comprise of (*model, basis*) pairs and are precomputed as follows: using a chosen basis, if a feature point in a model hashes into a bin, then the model and the basis are recorded in the bin. In the recognition phase, an arbitrary pair of feature points in the scene is chosen as a basis and the coordinates of the feature points in the scene are computed relative to this basis. The new coordinates are used

---

\*This research was performed as part of the MAARC (Models, Algorithms, and Architectures for Reconfigurable Computing) project. This work is supported by DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

to hash into the hash table. Votes are accumulated for the (*model, basis*) pairs stored in the hashed location. The pair winning the maximum number of votes is chosen as a candidate for matching. The execution of the recognition phase corresponding to a basis pair is termed as a *probe*.

There have been some prior efforts in parallelizing geometric hashing on HPC platforms [4, 16, 18]. A major problem in these implementations is that their performance degrades significantly due to the irregular communication in accessing the hash bins and in voting because the hash table and the vote boxes are distributed among the processing nodes. We refer to the congestion in accessing the bins as well as in voting as “memory congestion problems”.

Recently, configurable computing ideas [3, 8] have shown attractive speedups for many applications. They offer large scale parallelism and highly customized solutions. Field Programmable Gate Arrays (FPGAs) are becoming one of the major configurable computing devices which offer low development cost, rapid prototyping, and user controlled reconfigurability.

In this paper, we develop a design technique to parallelize the probe operation on an FPGA-based platform. We first transform the hash table which contains symbolic data into a “bit-level” representation. By regularizing the data flow and exploiting large scale bit-level parallelism in hardware, our design avoids the memory congestion problems. This leads to high performance. Since we employ a bit-level hash table, there is no hash bin access between the processing nodes although the hash table is distributed among the processing nodes. All operations are performed locally in each processing node except for finding a maximum value over the processing nodes. Furthermore, we parameterize the application as well as the device characteristics. Based on these parameters, we derive an area-time efficient design. The implementation is simplified using a modular approach.

We have synthesized our design using Xilinx 4062 devices. Using a clock rate of 10MHz, the execution time for the probe operation on a scene consisting of 256 feature points is estimated to be 1.65 *milliseconds* using 32 FPGAs and 128M bytes of memory. In this design, as in the earlier experiments, we assume that the model database has 1024 models and each model is represented using 16 feature points. For the sake of comparison, a parallel algorithm was implemented on a 32-node IBM SP-2. In our implementation, each processing node has the entire set of vote boxes to reduce the communication cost and to re-

duce memory congestion. However, the hash table was partitioned such that each hash bin is evenly distributed among the processing nodes. This balances the load on the processing nodes during the voting process. All operations are performed locally except for finding the global maximum. Using between 64 and 512M bytes of memory in each processing node operating at 66MHz, the execution time was about 40 *milliseconds*. In earlier implementations, the same probe operation was performed in 240 *milliseconds* on a 32K-node CM2 [16] and in 382 *milliseconds* on a 32-node CM5 [18].

The organization of the paper is as follows. In Section 2, configurable computing is briefly introduced. The geometric hashing technique is outlined in Section 3. Section 4 discusses parallelization of geometric hashing. In Section 5, implementation details are shown and the performance of the proposed design is compared with earlier results. Concluding remarks are made in Section 6.

## 2 Configurable Computing

Configurable computing has recently gained much attention with the promise of delivering an order of magnitude performance improvement over general-purpose processors. The paradigm of computing in space (i.e., a series of computations on several functional units), as opposed to computing in time (i.e., a series of computations executed in sequence on a single functional unit), is being actively explored. There are several directions in which research is being carried out to realize the potential of configurable computing.

The idea of a VLSI array of processors overlaid with a reconfigurable bus system, and an abstract model based on this architecture was proposed in [15]. Based on this initial work, several abstract models of reconfigurable architectures and fast parallel algorithms for many problems have been described in the literature. For example, efficient algorithms for fundamental data movement operations [15], sorting [11], and image processing [10] have been developed on the reconfigurable meshes. There have been several prototype implementations of such abstract models. Such architectures include Abacus [2] and YUPPIE [14].

Recently, the advent of Field Programmable Gate Arrays (FPGAs) has given rise to new opportunities in the configurable computing area. Traditionally, FPGAs have been used for rapid prototyping and emulation. The main bottleneck in using these devices as configurable computing engines has been the time for reconfiguration. Current and future generation devices such as CLAY, XC6200, DPGA etc. alleviate

the above problem by providing partial and dynamic reconfigurability [8]. In these devices, it is possible to modify the configuration of a part of the device while the configuration of the remaining part is unchanged. Some devices permit this partial reconfiguration even while other logic blocks are performing computations. Unlike such fine-grain devices, coarse grain devices in which multiple contexts of the configuration can be stored in the logic block and the context is dynamically switched have been proposed (for example, see [8]). Also, there are efforts under way to develop coupled architectures in which a reconfigurable array and a processor core cooperate on a computational task, exploiting the strengths of both architectures (for example, see [9]). Wormhole runtime reconfiguration has been proposed in [1]. In this approach, as the stream of data moves through the reconfigurable hardware, it rapidly creates and modifies datapaths and computing resources along the way. There have been some efforts to exploit dynamic reconfiguration [3, 13]. In these, the connections are configured based on the input data or the intermediate result of the computation.

Configurable computing provides the ability to redefine the hardware/software boundary in computing systems. This paradigm change results in new computation models, new programming methods, and new approaches to implementation of applications. Some of the greatest gains in this field may well come from providing appropriate abstractions of this technology to algorithm developers and compiler designers to allow them control over hardware that has not been previously exploited [13].

### 3 Object Recognition Using Geometric Hashing

In a model-based recognition system, a set of objects is given and the task is to find instances of these objects in a given scene. The objects are represented as sets of geometric features, such as points or lines, and their geometric relations are encoded using a minimal set of such features. The task becomes more complex if the objects overlap in the scene and/or other occluded unfamiliar objects exist in the scene.

Many model-based recognition systems are based on hypothesizing matches between scene features and model features, predicting new matches, and verifying or changing the hypotheses through a search process. Geometric hashing, introduced by Lamdan and Wolfson [12], offers a different approach. It can be used to recognize flat objects under weak perspec-

tive. Because of such robustness, geometric hashing has been employed in the DARPA next-generation, model-based ATR (Automatic Target Recognition) system [6]. In the following, for the sake of completeness, we briefly outline the geometric hashing technique. Additional details can be found in [12].

Figure 1 illustrates the geometric hashing algorithm. The algorithm consists of two procedures, *preprocessing* and *recognition*.

#### Preprocessing:

The preprocessing procedure is executed off-line and only once. In this procedure, the model features are encoded and are stored in a hash table. The information is stored in a highly redundant multiple-viewpoint way. Assume each model in the database has  $n$  feature points. For each ordered pair of feature points in the model chosen as a basis, the coordinates of all other points in the model are computed in the orthogonal coordinate frame defined by the basis pair. Then,  $(model, basis)$  pairs are entered into the hash table bins by applying a given hash function  $f$  to the transformed coordinates.

#### Recognition:

In the recognition procedure, a scene consisting of  $S$  feature points is given as input. An arbitrary ordered pair of feature points in the scene is chosen. Taking this pair as a basis, the coordinates of the remaining feature points are computed. Using the hash function on the transformed coordinates, a bin in the hash table (constructed in the preprocessing phase) is accessed. For every recorded  $(model, basis)$  pair in the bin, a vote is collected for that pair. The pair winning the maximum number of votes is taken as a matching candidate. The execution of the recognition phase corresponding to one basis pair is termed as *probe*. If no  $(model, basis)$  pair scores high enough, another basis from the scene is chosen and a probe is performed.

Note that, the basis set can be chosen as a set of single points, point pairs, or triple points depending on the required functionality for occlusion, rotation, translation, and perspective. The object features can also be represented by other geometric features such as *lines* [17].

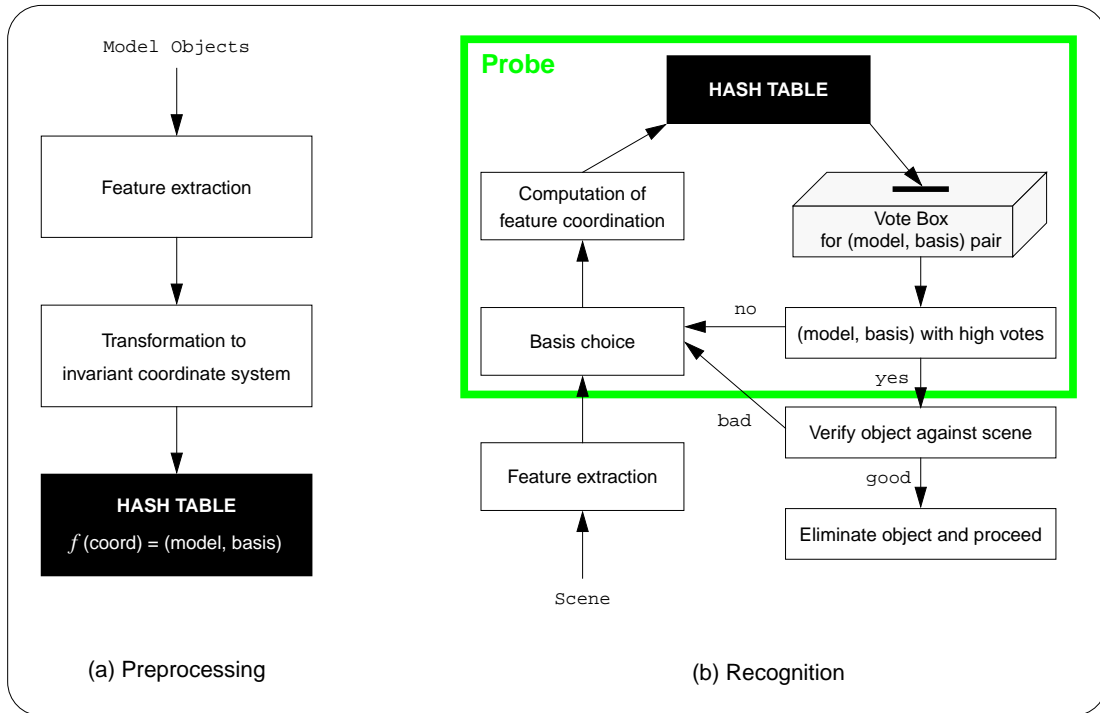


Figure 1: Illustration of the geometric hashing technique.

## 4 Parallel Geometric Hashing on an FPGA-based Configurable Computing Platform

In this section, we describe our parallel technique to implement the recognition phase. We first explain our bit-level hash table. Then, our parallel probe algorithm and its FPGA-based design are proposed. Finally, an analysis of our design is described.

We will not elaborate on parallelizing the preprocessing phase, since it is a one time process and can be carried out off-line. In the following, we ignore the initialization costs, such as loading the scene points to the processors and loading the hash table into the processor array. These assumptions were also made in the previous algorithms and in the implementations reported in [4, 16, 18].

The major difficulty in parallelizing the probe operation is that the performance depends on the partitioning and distribution of hash bins, the distribution of the votes generated, and the total number of votes generated. We refer to the congestion in accessing the bins as well as in voting as “memory congestion problems”.

There have been several prior efforts in parallelizing

the geometric hashing algorithms [4, 16, 18]. The implementations in [4, 16] have been performed on SIMD hypercube-based machines. A major problem in both the implementations is the requirement of large number of processing nodes. In [4], the number of processing nodes used is the same as the number of bins in the hash table. Thus,  $O(Mn^3)$  processing nodes are needed. In implementations reported [16, 18], the  $(\text{model}, \text{basis})$  entries in a hash bin were represented as a linked list. Note that, the number of such entries in each hash bin can vary over the hash bins. By partitioning the hash table and the vote boxes statically among the processing nodes, the memory congestion in bin access as well as in voting significantly degrades the performance. In [18], these problems were solved using a sort-based approach. This approach handles congestion in bin access as well as in voting. However, additional overhead caused in implementing such a technique makes it attractive only if the computational cost associated with accessing the hash bin and processing the generated votes is high.

### 4.1 Bit-level Hash Table

In this paper, we propose a simple memory structure which can be accessed in parallel without memory

congestion. By regularizing the data flow and exploiting a high degree of bit-level parallelism in hardware, large speedup is achieved. For the sake of explanation, we assume that there are no multiple entries of the same  $(model, basis)$  pair in a hash bin. Note that, the number of  $(model, basis)$  pairs recorded in a hash bin is upper bounded by  $\frac{Mn(n-1)}{2}$ .

The hash bin which has the linked list structure is converted to a bit-level hash bin. Figure 2 shows this hash table conversion. Let  $UID(model, basis)$  denote an unique number, between 1 and  $\frac{Mn(n-1)}{2}$ , assigned to each  $(model, basis)$  pair. Then, each hash bin is converted into our “bit-level” representation of size  $\frac{Mn(n-1)}{2}$  bits as follows.

1. Initialize each of the  $\frac{Mn(n-1)}{2}$  locations to “0”.
2. For each  $(model, basis)$  pair recorded in a hash bin, enter a “1” in the location  $UID(model, basis)$ .

The corresponding  $(model, basis)$  pair in the hash table is marked as ‘1’ in the bit-level hash table. Thus, a hash table in which the number of entries for each hash bin may not be uniformly distributed across the hash bins is mapped to a bit-level hash table having a regular structure. Using this bit-level hash table, we can exploit a high degree of parallelism and eliminate the congestion in hash bin access.

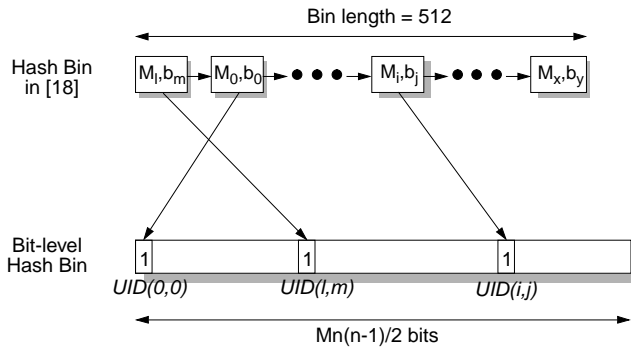


Figure 2: Hash table conversion.

## 4.2 Parallel Probe Algorithm

The basic strategy of our design is to access the  $\frac{Mn(n-1)}{2}$  bit locations in a hash bin in parallel and then update the corresponding vote boxes in parallel (See Figure 3(a)). Thus, we can perform a probe operation without any memory congestion. Note that a single FPGA chip may not have enough number of

Combinational Logic Blocks (CLBs) to handle the bit streams in parallel. Therefore, multiple FPGA chips are required. Finding the maximum among the vote boxes distributed in multiple FPGA chips is performed in 2 steps: find the *local maximum* in each FPGA chip and then find the *global maximum* across the FPGA chips.

To obtain a modular design, we partition our design into three modules: *pre-processing module*, *main-processing module*, and *post-processing module*. The pre-processing module generates the bin address of the bit-level hash table. Given  $(x, y)$ , the coordinates of a scene point, the co-ord transformer first converts it into basis-relative coordinates  $(u, v)$ . Then, the bin address generator converts  $(u, v)$  into a corresponding hash bin address. These two operations can be easily implemented using table look-up even though they involve complex arithmetic operations. Then, the main-processing module accesses the hash table using the computed bin address and detects *local* maximums in each FPGA chip. Finally, a *global* maximum across the FPGA chips is detected by the post-processing module using a comparator tree (See Figure 3(b)). Since parallelism is exploited in the main-processing module, in the following, we focus on designing that module.

A basic unit for the main-processing module consists of a pair of FPGA and local memory modules, and is denoted as *Processing Element* (PE). In general, trade-offs between area and time are possible when a specific function is implemented in hardware. Especially, the available sizes of Commercial Off-The-Shelf (COTS) devices may not match well with the basic unit of our design. Thus, the *virtual* PEs in our design need to be mapped onto *physical* PEs which can be implemented using COTS devices. To derive an area-time efficient design using COTS devices, we define parameters which characterize our design with respect to area and time (See Figure 4). Using  $P$  PEs, our design accesses  $Pn$  bits of the hash table in parallel. Since  $\frac{Mn(n-1)}{2}$  bits need to be accessed in each hash bin and since typically,  $\frac{Mn(n-1)}{2} > Pn$ , the number of time multiplexing required to map virtual PEs to physical PEs is  $\lceil t = \frac{Mn(n-1)}{2Pn} \rceil$ . Details of the time multiplexing to realize an area-time efficient design using COTS devices are described in the next section.

## 4.3 Design Analysis

The execution time for a probe using  $P$  PEs can be analyzed as follows. Throughout this paper, we assume that a memory access and an arithmetic or logic operation (such as **ADD**, **MUX**, **COMPARE**) can be

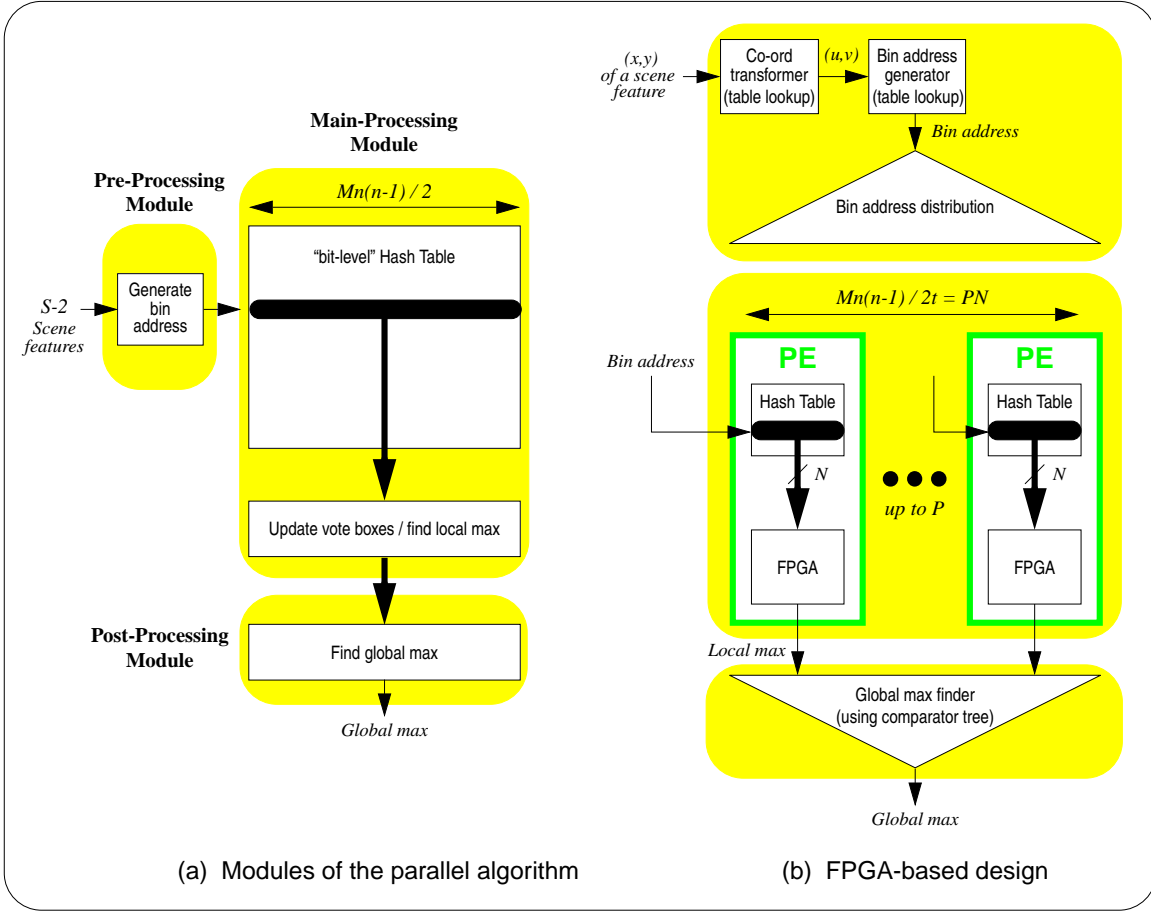


Figure 3: Parallel probe algorithm and its FPGA-based design.

performed in unit time. The hash bin address can be generated in  $O(1)$  times using table look-up. The generated hash bin address can be distributed to  $P$  PEs in  $O(\log P^1)$  time. The hash bin accesses for  $S - 2$  feature points can be performed concurrently with the operation to detect local maximums from the preceding hash bin accesses. Thus, the time to detect the local maximums over  $t$  multiplexing is  $(S - 2) \times t + \log N = O(\frac{SMn^2}{PN} + \log N)$ . The hash bin access and the operation to detect a local maximum are pipelined. The final operation to detect a global maximum can be performed in  $O(\log P)$  time.

**Theorem 1** *Given a model database having  $M$  models where each model is represented using  $n$  feature points, a probe on a scene consisting of  $S$  feature points can be performed in  $O(\frac{SMn^2}{PN} + \log N + \log P)$*

<sup>1</sup>All logarithms in this paper are to base 2.

time on an FPGA-based platform having  $P$  PEs,  $1 < P \leq \frac{Mn(n-1)}{2N}$ , where  $N$  denotes the width of FPGA-memory datapath in a PE.

## 5 Implementation Details and Performance Estimate

In this section, we first discuss various issues in implementing the design technique developed in Section 4 on an FPGA-based platform. Then, we describe a design using Xilinx 4062 FPGA devices. Our design is motivated to achieve large speedup for typical size of images and models used by the vision community. We have chosen not to perform device dependent optimizations to improve performance. Figure 5 shows our development environment. We have synthesized our design using Synopsis FPGA compiler. Place and route was performed using Xilinx tools (XACT Devel-

		Description
Application Parameters	M	Number of model objects
	n	Number of feature points in a model object
	S	Number of feature points in a scene
Design Parameters	P	Number of PEs required
	N	Width of FPGA-memory datapath in a PE
	t	Number of time multiplexing required to map virtual PEs to physical PEs

Figure 4: Parameters used for deriving an area-time efficient design.

opment System) to create a configuration file for an FPGA on a Sun Ultra Enterprise. Then, configuration file is downloaded onto the FPGA development board which consists of FPGAs and memory modules. The board is connected to a PC through PCI Local Bus.

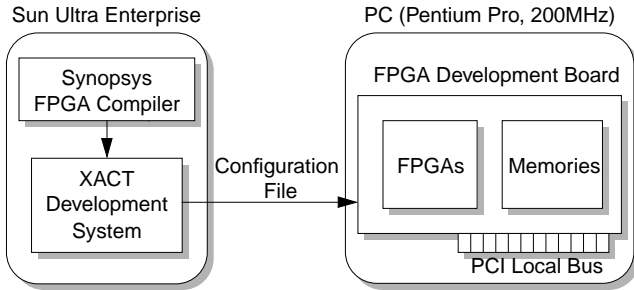


Figure 5: Our development environment

For the sake of illustration and evaluation of the resulting design, we consider a typical scenario as follows: We used a synthesized model database containing 1024 models. Each model consists of 16 randomly generated feature points in 2 dimensions. This results in a hash table having  $4 \times 2^{20}$  entries. These feature points were generated according to a Gaussian distribution with zero mean and unit standard deviation as in [16, 18]. Similarly, 256 scene points were synthesized using a normal distribution. The equalization technique in [16, 18] was applied to quantize the transformed coordinates, *i.e.*, for each transformed point  $(u, v)$ , the following hash function (where  $\sigma$  denotes the standard deviation of the set of model points) is applied [18]:

$$f(u, v) = (1 - e^{-\frac{u^2+v^2}{3\sigma^2}}, \text{atan2}(v, u))$$

According to constraints imposed by Commercial

Off-The-Shelf (COTS) devices, we first determine both the structure of each PE and the configuration of the PEs to synthesize an area-time efficient solution. Based on the configuration of the PEs in the main-processing module, the logic for the pre-processing and post-processing modules are determined.

### 5.1 Design Trade-offs

To illustrate feasibility of implementation and demonstrate resulting speedup, we assume that Xilinx 4062 FPGA chips and  $512K \times 32$  bit memory modules are used for the implementation. The scenario considered above results in a hash table having  $8K$  hash bins [16, 18]. To represent the same hash table using our bit-level design, we need  $8K \times 120K$  bits (See Figure 6 (a)). For the sake of explanation, let  $MEM$  denote the number of memory modules needed to store the hash table. To implement this bit-level hash table with commercially available  $512K \times 32$  bit memory modules,  $MEM$ ,  $N$ , and  $t$  must satisfy the following constraints:

- $512K \times 32 \text{ bits} \times MEM \geq 8K \times 120K \text{ bits}$ ,
- $\frac{512K}{t} \geq 8K$ , and
- $32 \text{ bits} \times MEM \geq \frac{120K \text{ bits}}{t} \geq PN \text{ bits}$ .

We can implement 64 vote boxes and the logic to find a local maximum among the vote boxes in an FPGA. Thus, the width of FPGA-memory datapath,  $N$ , becomes 64 and this amount of parallelism is achieved in a PE.

From the above constraints, a feasible configuration is  $MEM = 60$ ,  $t = 64$ , and  $P = 30$ . Designs with large  $P$  ( $\geq 30$ ) are not area efficient due to large number of memory modules. Also, note that the number of memory modules is 60 and the number of PEs is 30. Thus, two  $512K \times 32$  bit memory modules are assigned to each PE to support 64-bit parallelism.

Figure 6 shows the actual mapping of the hash table onto the memory modules in our design. In this example, the size of the bit-level hash table is  $8K \times 120K$  bits. A smaller *sub-table* whose size is  $8K \times 4K$  bits is assigned to a PE (See Figure 6 (b)). Since the width of a hash bin in the sub-table is  $4K$  bits and  $N = 64$  bits can be read from the sub-table simultaneously, one hash bin access results in 64 reads to the memory module. When we map the sub-table into an actual memory module, a column major order is used. Thus, the first  $8K \times 64$  bits are placed in the beginning of the memory module. The next  $8K \times 64$  bits are placed immediately after this (See Figure 6 (c)). Figure 6 (c).

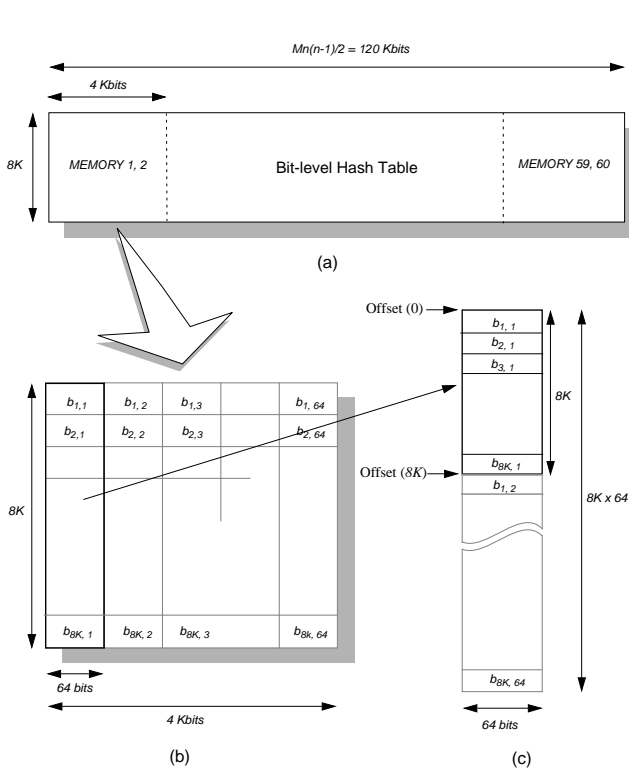


Figure 6: Memory organization. (a) Bit-level hash table, (b) Memory module (sub-table), and (c) The sequence of hash bins in a memory module.

Some details of the FPGA implementation of a PE is shown in Figure 7. It consists of 64 vote boxes, eight 8-to-1 multiplexers, and the logic to find a local maximum. The logic to find the local maximum consists of 8-input comparator tree and logic to update the local maximum. When a hash bin address is generated, 64 bits of the hash bin are fed into FPGAs from the memory modules. The corresponding vote boxes

count the number of '1's. Once the voting operation is completed, the votes are stored in registers and are multiplexed to be fed into the comparator tree. Using the comparator tree, a maximum is found and is compared with the previous local maximum. If necessary, the local maximum is updated. In the design of the PE, we can also use a 64-input comparator tree to find the local maximum directly from 64 vote boxes. Since the execution time for the voting operation is longer than that for finding the local maximum, we multiplex the votes in the registers and feed them to a smaller comparator tree. Thus, we can reduce the amount of logic to find the local maximum.

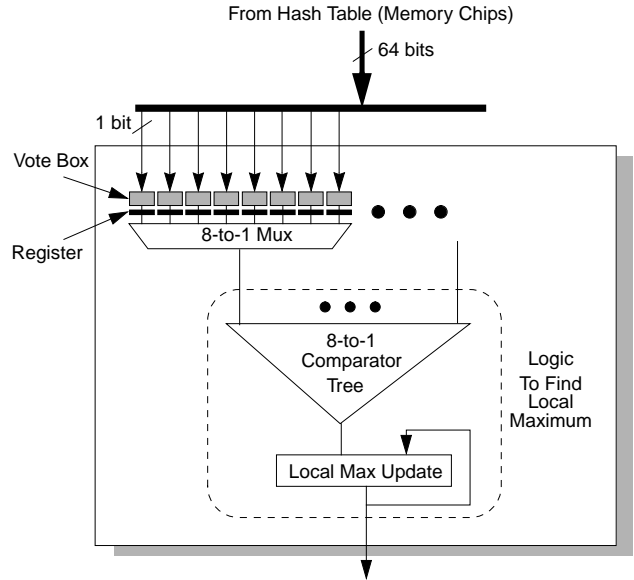


Figure 7: FPGA-based implementation of a PE.

For the above configuration, the logic for finding a maximum globally across the 30 PEs can be implemented in a single FPGA. Also, the generation of the transformed co-ordinates as well as hashing the addresses can be performed using 1 FPGA chip by using table look-up. In the above design using 30 PEs, since the total number of PEs is relatively small, the generated hash bin address is distributed directly over a bus to each PE rather than using a tree topology.

## 5.2 Performance Estimate

Our FPGA-based design has been developed using VHDL and synthesized using Synopsys synthesis tools to generate a gate-level design. Then, Xilinx development tools were used for placing and routing our design using FPGAs. Since all the PEs are identical

	Platform	No of Scene Points	Execution Time
Bourdon [4]	8K-node CM2	*	2000-3000 msec
Rigoutsos [16]	32K-node CM2	200	240 msec
Wang [18]	32-node CM5	256	382 msec
Our Parallel Algorithm	32-node SP-2	256	40 msec
FPGA-based design	32 PEs	256	1.65 msec

\* Two real life models (a key and a shaver having 15 feature points and 14 feature points, respectively) and a scene consisting of 28 feature points were used.

Figure 8: Comparison of our FPGA-based implementation with previous parallel implementations on HPC platforms

except for the contents of memory, we synthesized a PE.

Our design runs at a clock rate of 10MHz and the estimated execution time for a probe is 1.65 *milliseconds* using 32 PEs. Each PE consists of an FPGA and a local memory. Two additional FPGA chips and memory modules are required to implement the pre- and post-processing modules.

For the sake of comparison, a sequential algorithm has been implemented using C. On an UltraSPARC Model 140 (143MHz clock, 128M byte memory, 7.44 *SPECint\_95*, 10.40 *SPECfp\_95*), it takes about 300 *milliseconds* to perform a probe operation. We implemented a parallel algorithm on a 32-node IBM SP-2. Each node of SP-2 had 66MHz processors and 64 up to 512M bytes of memory. The performance benchmarks of the processors were 3.14 *SPECint\_95* and 7.50 *SPECfp\_95*. In our parallel algorithm, we evenly distribute the hash table (which is vertically partitioned) to 32 processing nodes. However, unlike the previous implementations [18], each processing node has the complete set of vote boxes. Initially, all scene points are sent to each processing node. Each node performs a voting operation locally using the distributed hash table. The results of voting are sent to other processing nodes using an “all-to-all” communication so that all the data corresponding to a vote box is combined and stored in a single PE. Based on the collected votes, local maximums are computed in each processing node and a global maximum is computed over the 32 processing nodes. The implementation on SP-2 was performed using MPI. The execution time was about 40 *milliseconds*. Using one processing node of SP-2, the execution time was about 500

*milliseconds*. Therefore, our FPGA-based solution can achieve a speedup of close to 176 and 24, respectively. A comparison with parallel implementations on HPC platforms is shown in Figure 8.

Note that, the memory requirement per PE in our design is only 4M bytes, whereas the size of local memory in each node of SP-2 is between 64 and 512M bytes. Also, our design assumes 10MHz clock while the processors in SP-2 run at 66MHz.

Although we use 32 PEs, our design is scalable. Note that, as the number of PEs increases, the required number of time multiplexing decreases. However, the internal structure of each PE, such as the width of FPGA-memory datapath, is not affected by the number of PEs used. The upper limit on the number of PEs is obtained by setting  $t = 1$  (no time multiplexing). As we mentioned earlier, however, designs with large  $P$  are not area efficient due to the number of memory modules used and low FPGA utilization.

Previously, we have assumed that in any hash bin there is no more than one (*model, basis*) pair. Thus, we need only one bit to mark  $UID(model, basis)$  in our bit-level hash bin. However, if there are more than one identical (*model, basis*) pair in a hash bin, then our design can be easily modified to handle this. To allow  $K$  identical (*model, basis*) pairs,  $\lceil \log K \rceil$  bits are needed for each  $UID(model, basis)$  in a bit-level hash bin. The memory size of the hash table also increases by a factor of  $\lceil \log K \rceil$ . To generate the bit-level hash bin, the number of identical (*model, basis*) pairs is counted and is stored in the corresponding  $UID(model, basis)$  location. The design for vote boxes needs to be modified. An additional  $\lceil \log K \rceil$ -bit adder is required for each vote box.

## 6 Conclusion

We have shown an area-time efficient FPGA-based design for the probe step in geometric hashing. In our design, we first transform a hash table which contains symbolic data into a bit-level representation. By regularizing the data flow and exploiting bit-level parallelism in hardware, our design avoids memory congestion. In addition, the implementation is simplified using a modular approach.

Performance estimates are very encouraging. Given a model database having 1024 models where each model is represented using 16 feature points, a probe operation on a scene consisting of 256 feature points can be performed in 1.65 *milliseconds* on an FPGA-based platform (32 FPGAs and 128M bytes of memory). This result does not assume any distribution of hash bin lengths or scene points. For the same probe

operation, a parallel algorithm on a 32-node IBM SP-2 required 40 *milliseconds*, and the earlier implementation required 240 *milliseconds* on a 32K-node CM2 and 382 *milliseconds* on a 32-node CM5.

The work reported here is part of the USC MAARC (Models, Algorithms, and Architectures for Reconfigurable Computing) project for algorithmic configurable computing [13]. In this project, characteristics of state-of-the-art configurable hardware are abstracted to capture their capabilities. Using such representations of configurable devices, system-level models that allow the development of new algorithms for mapping applications on configurable systems are formulated. Using the models and metrics for configurable computing, high-performance algorithms for these architectures are designed.

## Acknowledgment

We would like to thank Steve Casselman, David Franklin, and John Schewel of the Virtual Computer Corporation for their assistance in evaluating our designs using their development boards.

## References

- [1] R. Bittner and P. Athanas, "Wormhole Run-Time Reconfiguration," *International Symposium on Field Programmable Gate Arrays (FPGA'97)*, February 1997.
- [2] M. Bolotski, R. Amirtharajah, W. Chen, T. Kutscha, T. Simon, and T. Knight Jr., "Abacus: A High-Performance Architecture for Vision," *International Conference on Pattern Recognition*, 1994.
- [3] K. Bondalapati and V. K. Prasanna, "Reconfigurable Meshes: Theory and Practice," *Workshop on Reconfigurable Architectures, IPPS'97*, 1997.
- [4] O. Bourdon and G. Medioni, "Object Recognition Using Geometric Hashing on the Connection Machine," *International Conference on Pattern Recognition*, vol. 2, pp. 596-600, 1990.
- [5] Y. Chung, "Asynchronous Parallel Algorithms for Irregular Vision Problems on Distributed Memory Machines," *Ph.D. Thesis, University of Southern California*, August 1997.
- [6] DARPA, "Moving and Stationary Target Acquisition and Recognition (MSTAR) Program," <http://yorktown.dc.isx.com/iso/battle/mstar.html>.
- [7] A. Dandalis and V. K. Prasanna, "Fast Parallel Implementation on DFT using Configurable devices," *7th International Workshop on Field-Programmable Logic and Applications*, 1997.
- [8] A. DeHon, "Reconfigurable Architectures for General Purpose Computing," *Technical Report, MIT AI Lab*, September 1996.
- [9] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, April 1997.
- [10] J. Jang, H. Park, and V. K. Prasanna, "A Fast Algorithm for Computing the Histogram on Reconfigurable Mesh," *Frontiers of Massively Parallel Computation*, pp. 244-251, October 1992.
- [11] J. Jang and V. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Journal of Parallel and Distributed Computing*, vol. 25, pp. 31-41, 1995.
- [12] Y. Lamdan and H. Wolfson, "Geometric Hashing: A General and Efficient Model Based Recognition Scheme," *International Conference on Computer Vision*, pp. 238-249, 1988.
- [13] MAARC Homepage, <http://maarc.usc.edu>.
- [14] M. Maresca and H. Li, "Connection Autonomy in SIMD Computer: a VLSI Implementation," *Journal of Parallel and Distributed Computing*, vol. 7, pp. 302-320, 1989.
- [15] R. Miller, V. Prasanna Kumar, D. Reisis, and Q. Stout, "Meshes with reconfigurable Buses," *5th MIT Conference on Advanced Research in VLSI*, pp. 163-178, March 1988.
- [16] I. Rigoutsos and R. Hummel, "Massively Parallel Model Matching: Geometric Hashing on the Connection Machine," *IEEE Computer*, pp. 33-42, 1992.
- [17] F. Tsai, "Using Line Features for Object Recognition by Geometric Hashing," *Technical Report, New York University*, 1993.
- [18] C. Wang, V. K. Prasanna, H. Kim, and A. Khokhar, "Scalable Data Parallel Implementations of Object Recognition Using Geometric Hashing," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 96-109, 1994.