PAPER *Special Issue on Reconfigurable Computing*

An Image Retrieval System Using FPGAs

Koji NAKANO[†], ***Regular Member and*** **Etsuko TAKAMICHI**[†], ***Student Member***

SUMMARY The main contribution of this paper is to present an image retrieval system using FPGAs. Given a template image T and a database of a number of images $I_1, I_2, \ldots$, our system lists all images that contain a subimage similar to T. More specifically, a hardware generator in our system creates the Verilog HDL source of a hardware that determines whether I_i has a similar subimage to T for any image I_i and a particular template T. The created Verilog HDL source is compiled and embedded in an FPGA using the design tool provided by the FPGA vendor. Since the hardware embedded in the FPGA is designed for a particular template T, it is an instance-specific hardware that allows us to achieve extreme acceleration. We evaluate the performance of our image matching hardware using a PCI-connected Xilinx FPGA and a timing analyzer. Since the generated hardware attains up to 3000 speed-up factor over the software solution, our approach is promising.

key words: *FPGA-based computation, image matching, instance-specific hardware*

1. Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware designed by users can be embedded instantly. Typical FPGAs consist of an array of programmable logic blocks, memory blocks, and programmable interconnect between them. The logic block usually contains a four-input logic function and/or several flip-flops. The memory block is usually a dual-port RAM which can be read/written a word of data for distinct addresses in the same time. Using design tools provided by FPGA vendors, a hardware logic designed by users can be embedded in the FPGA.

Our basic idea for accelerating computations using the FPGAs is inspired by the notion of *the partial computation* [2], [4]. Let $f(x, y)$ be a function that we have to evaluate to solve a problem. Sometimes, a function $f(x, y)$ is repeatedly evaluated only for a fixed x. If this is the case, the computation of $f(x, y)$ can be simplified by evaluating an instance-specific function f_x such that $f_x(y) = f(x, y)$. For instance, imagine a problem such that an algorithm to solve it evaluates $f(x, y) = x^3 + x^2y + y$ repeatedly. If $f(x, y)$ is evaluated only for $x = 2$, then we can simplify the formula such that $f_2(y) = 8 + 5y$. Actually, it is known [10] that the multiplication of two integers can be done efficiently if one of the integers is fixed. The optimization of function f_x for a given particular x is called the partial computation.

Usually, the partial computation has been used for optimizing a function f_x in the context of software, i.e., sequential programs [2], [4]. Our novel idea is to built a hardware that is optimized to compute $f_x(y)$ for a fixed x and various y's. More specifically, our goal is to present an FPGA-based instant-specific solution for a type of problems that involves a function evaluation for $f(x, y)$ satisfying the following properties:

- the value of a fixed instance x depends on the instance of the problem, and
- the value of $f(x, y)$ is evaluated repeatedly for various y to solve the problem.

The FPGA-based instance-specific solution that we propose evaluates $f_x(y)$ $(= f(x, y))$ using a hardware for function f_x. If a problem we need to solve satisfies these properties, it is worth attempting the instance-specific solution. For example, for a fixed context free grammar G, let $f_G(w)$ $(= f(G, w))$ be a function that determines if G derives a string w. It is known that an FPGA-based instance-specific solution for $f_G(w)$ is much faster than the software solution [1].

Suppose that an image database $\mathcal{I}$ containing a number of gray-scale images $\{I_1, I_2, \ldots\}$ and a template image T are given. We assume that T is small, say, 32×32 or 64×64 while each I_i is large, say, 1024×1024 or larger. We are interested in the task of listing all images in $\mathcal{I}$ that contains a similar subimage to T. This task has many applications in the areas such as object recognition, and vehicle tracking [3]. The main contribution of this paper is to present an FPGA-based instance-specific hardware solution for this task. More precisely, let $D(T, I_i)$ denote a function that returns a value indicating the difference between T and I_i such that the value of $D(T, I_i)$ is small if I_i has similar subimage to T. Our idea is to embed a hardware that computes $D_T(I_i)$ $(= D(T, I_i))$ in a PCI-connected FPGA. We have developed a system illustrated in Fig. 1 that computes $D_T(I_1), D_T(I_2), \ldots$ using the FPGA. Given a template image T, our hardware generator automatically creates a Verilog HDL source program which is designed for computing $D_T(I)$. More precisely, the generator is written in a C-language, which generates a Verilog HDL source program for a

Manuscript received September 6, 2002.

Manuscript revised November 18, 2002.

[†]The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology, Ishikawa-ken, 923–1292 Japan.

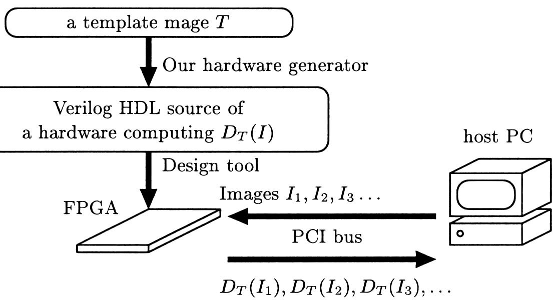

a template mage T
Our hardware generator
Verilog HDL source of a hardware computing $D_T(I)$
Design tool
host PC
FPGA
Images $I_1, I_2, I_3 \dots$
PCI bus
$D_T(I_1), D_T(I_2), D_T(I_3), \dots$

Fig. 1 Our FPGA-based image retrieval system.

matrix of binary image T in a few seconds. The source program is complied using a design tool provided by an FPGA vendor. The created hardware is embedded in the PCI-connected FPGA. The host PC sends images $I_1, I_2, \dots$ stored in an image database $\mathcal{I}$ to the PCI-connected FPGA. The FPGA computes the values $D_T(I_i)$ in turn, and returns each of them to the host PC. The host PC lists the images whose $D_T(I_i)$ is no larger than the threshold value. Although the time necessary to compile the Verilog HDL source and embedded into the FPGA is very long, say several hours, the total computing time can be decreased if database $\mathcal{I}$ has a large number of images.

Let T be a template image with $m \times m$ pixels and I be an image with $n \times n$ pixels. We assume that T has e effective pixels ($e \leqq m^2$) that are taken into account for image matching. As we are going to show later, the evaluation of $D(T, I)$ for L-level gray-scale images takes $O(n^2 e)$ time by a software solution. Our instance-specific FPGA solution runs in $\frac{n^2}{m}$ local clock cycles using $O(me)$ gates, $O(m^2)$ flip-flops, and an $(m-1)n$-bit block RAM for binary images, and in $\frac{n^2 \log L}{m}$ local clock cycles using $O(me)$ gates, $O(m^2 \log L)$ flip-flops, and $\log L$ block RAMs with mn bits each for L-level gray-scale images. Thus, from the theoretical point of view, our FPGA-based instance specific solution is much faster than the conventional software solution.

We have evaluated the performance of our hardware using a timing analyzer for the Xilinx VirtexII series FPGA XC2V8000 (Speed Grade 4). Further, since XC2V8000 is not available, we have tested our hardware using a smaller FPGA, Spartan2 (XC2S150, Speed Grade 6) PCI card Strathnuey [6]. Since the generated hardware attains up to 3000 speed-up factor over the sequential algorithm, our approach is a promising solution. An image matching hardware using an FPGA has been proposed [5]. Their hardware is not instance-specific, does not support gray-scale images, and runs in $O(n^2)$ clock cycles. Thus, our hardware is a significant improvement on the FPGA-based image matching hardware.

This paper is organized as follows. Section 2 defines the function $D_T(I)$ for a template image T and an image I formally, and evaluate the computing time by a software solution. In Sect. 3, we show a hardware that computes $D_T(I)$ for binary images and evaluate the performance. This hardware runs in less than n^2 local clock cycles. In Sect. 4, we parallelize the hardware presented in Sect. 3. The parallelized version of our hardware runs in $\frac{n^2}{m}$ local clock cycles. Section 5 presents an image matching hardware for L-level gray-scale images. This hardware runs in $\frac{n^2 \log L}{m}$ local clock cycles. Section 6 evaluates the performance of our hardware and compare with the software solution. Section 7 offers concluding remarks.

2. The Image Difference Function

The main purpose of this section is to define *the difference function* $D(T, I)$ for a template image T and an image I to clarify our work presented in this paper.

An L-level gray-scale image I of size $m \times m$ is an $m \times m$ two dimensional array with each element (or pixel) taking an integer in the range $[0, L-1]$. The value of an (i, j) pixel $I_{i,j}$ ($1 \leqq i, j \leqq m$) of I corresponds to its intensity. In other words, pixel (i, j) is black if $I_{i,j} = 0$ and white if $I_{i,j} = L - 1$. We assume that pixel $(1, 1)$ is the top of the leftmost column of I. An L-level gray-scale image I is a *binary image* if $L = 2$. An $m \times m$ template image T is an image with "don't care", that is, an $m \times m$ two dimensional array with each element taking either an integer in $[0, L-1]$ or a special value d. An (i, j) pixel ($1 \leqq i, j \leqq m$) of T is "don't care" if $T_{i,j} = d$. Let e denote the number of *effective pixels*, which are non-"don't care" pixels in T. The value of e, which depends on the applications, can be much smaller than m^2.

Let D be the function that returns an integer for a template image T and an image I such that

$$D(T, I) = \sum_{T_{i,j} \neq d} |T_{i,j} - I_{i,j}|. \tag{1}$$

Intuitively, $D(T, I)$ is the sum of the difference of the brightness over all effective pixels. Clearly, $D(T, I)$ takes a larger value if they are less similar. Note that, for a binary template T and a binary image I, their difference is

$$D(T, I) = \sum_{T_{i,j} \neq d} T_{i,j} \oplus I_{i,j}, \tag{2}$$

where $\oplus$ denotes the exclusive OR operator.

Suppose that an image I is larger than a template image T. Let $n \times n$ ($n > m$) be the size of image I. Further, let $I[x, y]$ ($1 \leqq x, y \leqq n - m + 1$) denote an $m \times m$ subimage of I that includes all pixels $I_{i',j'}$ ($x \leqq i' \leqq x + m - 1$ and $y \leqq j' \leqq y + m - 1$). *The image difference function* $D(T, I)$ between a template T and an image I is

$$D(T, I) = \min_{1 \leqq x, y \leqq n-m+1} D(T, I[x, y]). \tag{3}$$

Clearly, $D(T, I)$ is small if I has a similar subimage to T. Also, let D_T denote a function such that $D_T(I) = D(T, I)$.

By evaluating $D(T, I_1), D(T, I_2), \ldots$ in turn, we can retrieve all images in a database of images $I_1, I_2, \ldots$, which have a similar subimage to T. Our goal is to accelerate the computation of evaluating $D(T, I)$. For later reference, let us evaluate the computing time necessary to compute $D(T, I)$ by a software (or a sequential algorithm). For an $m \times m$ template image T with e effective pixels and a subimage $I[x, y]$, the value of $D(T, I[x, y])$ can be computed in $O(e)$ time. Hence, the evaluation of $D(T, I[x, y])$ for all $I[x, y]$ $(1 \leqq x, y \leqq n-m+1)$ takes $(n-m+1)^2 \times O(e) = O(n^2 e)$ time. Therefore, the task of computing the image difference $D(T, I)$ takes $O(n^2 e)$ time. In the following sections, we will show that our FPGA-based instance-specific solution can perform this task for L-level gray-scale images in $\frac{n^2 \log L}{m}$ clock cycles.

3. An Image Matching Hardware for Binary Image Retrieval

In this section, we are going to show our FPGA-based instance-specific hardware that computes $D_T(I)$ $(= D(T, I))$ for a fixed template T and various images I. We start with a binary template T and a binary image I. We then go on to extend our hardware to support gray-scale images later.

Figure 2 illustrates our hardware for $m = 4$ that evaluates $D(T, I)$ using formulas (2) and (3). For simplicity, we assume that, for a template T of size $m \times m$, m pixels of image I can be supplied via PCI-bus in every local clock cycle. The frequency of local clock depends on the hardware design as we are going to show in Sect. 6. If the size of a template image T is larger than the PCI bit-width, the necessary bits can be sent in several clock cycles. Actually, 32-bit or 64-bit PCI buses with frequency 33 MHz or 66 MHz are widely used [9].

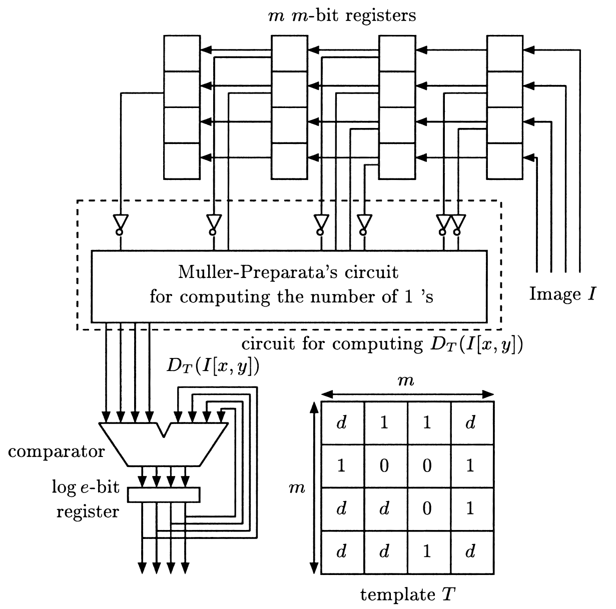
Next, we are going to show how our hardware works. Let $I_m[i, j]$ denote the set of vertical adjacent m pixels $I_{i,j}$, $I_{i+1,j}$, …, $I_{i+m-1,j}$. Note that the m pixels in any $I_m[i, j]$ can be transferred to the register in a local clock cycle. Every pixel in image I of size $n \times n$ is transferred to the m m-bit registers in $n \cdot (n - m + 1)$ clock cycles as follows:

```
for $i \leftarrow 1$ to $n - m + 1$ do
  for $j \leftarrow 1$ to $n$ do
    Perform the following two operations in parallel:
    1. $I_m[i, j]$ is transfered to the rightmost register;
    2. the $m$ registers are shifted to the left by one.
```

It should be clear that, subimage $I[1, 1]$ is stored in the registers when $(i, j) = (1, m)$. Further, every subimage $I[x, y]$ $(1 \leqq x, y \leqq n - m + 1)$ is stored in the registers when $i = x$ and $j = y + m - 1$.

We use a combinatorial circuit that computes $D_T(I[x, y])$ for subimage $I[x, y]$ currently stored in the m m-bit registers. It consists of parallel inversions and the Muller-Preparata's circuit [7], [8] that computes the number 1's in the input bits. For every pixel of template image T, the corresponding register bit or its inversion is connected to the Muller-Preparata's circuit if it is 1 or 0, respectively. Since T has e effective pixels, the Muller-Preparata's circuit computes the sum of e bits, which is equal to the value of $D_T(I[x, y])$. For the readers benefit, Fig. 3 illustrates the Muller-Preparata's circuit that computes the number of 1's in eight bits, where FA and HA denote the full adder and the half adder, respectively. It outputs the binary representation of the number of 1's. The reader should have no difficulty to confirm that it computes the number of 1's.

To compute the minimum of $D_T(I[x, y])$ over all x and y, a comparator and a $\log e$-bit register is used. The comparator computes the minimum of two $\log e$-bit integers. The register is storing the temporary min-


m m-bit registers
Muller-Preparata's circuit for computing the number of 1 's
Image I
circuit for computing $D_T(I[x,y])$
$D_T(I[x,y])$
comparator
$\log e$-bit register
m
m
template T

d	1	1	d
1	0	0	1
d	d	0	1
d	d	1	d

Fig. 2 A hardware implementation of our circuit computing $D_T(I)$.

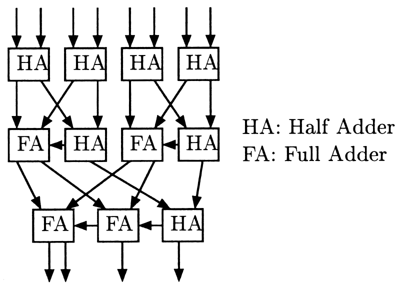


Fig. 3 The Muller-Preparata's circuit for computing the number of 1's in eight bits.

imum value of $D_T(I[x,y])$ so far. If the current value of $D_T(I[x,y])$ is smaller, then it is stored in the register. It should be clear that, after every pixel in image I is supplied to this circuit, the $\log e$-bit register stores $D_T(I)$.

Next, let us evaluate the performance and the hardware resources used by our hardware. As we discussed, our hardware computes $D_T(I)$ in less than n^2 clock cycles. The Muller-Preparata's circuit [7] that counts the number of 1's in e bits has $O(e)$ gates. Further, the $\log e$-bit comparator has no more than $O(\log e)$ gates. The m m-bit registers uses m^2 flips-flops and the $\log e$-bit register uses $\log e$ ($< 2\log m$) flip-flops. Thus, our hardware uses $O(e)$ gates and $O(m^2)$ flip-flops.

4. Parallel Image Matching for Binary Images

This section is devoted to show our parallel image matching architecture for further acceleration.

Figure 4 illustrates a part of our parallel image matching hardware. In order to reduce the number of clock cycles, we use m circuits computing $D_T(I[x,y]), D_T(I[x+1,y]), \ldots, D_T(I[x+m-1,y])$ in parallel. Note that, $D_T(I[x,y]), D_T(I[x+1,y]), \ldots, D_T(I[x+m-1,y])$ combined correspond to a subimage with $(2m-1)\times m$ pixels. Thus, we use m registers with $(2m-1)$ bits each to store a subimage of $(2m-1)\times m$ pixels. Again, we assume that m pixels in an image I are supplied in every local clock cycle. Hence, vertical $(2m-1)$ pixels cannot be transferred to the rightmost $(2m-1)$-bit register in every local clock cycle. To supply the $(2m-1)$ pixels to the register in every local clock cycle, we use an $(m-1)\times n$-bit cache, that is, a cache with $(m-1)$-bit data and $\log n$-bit address.

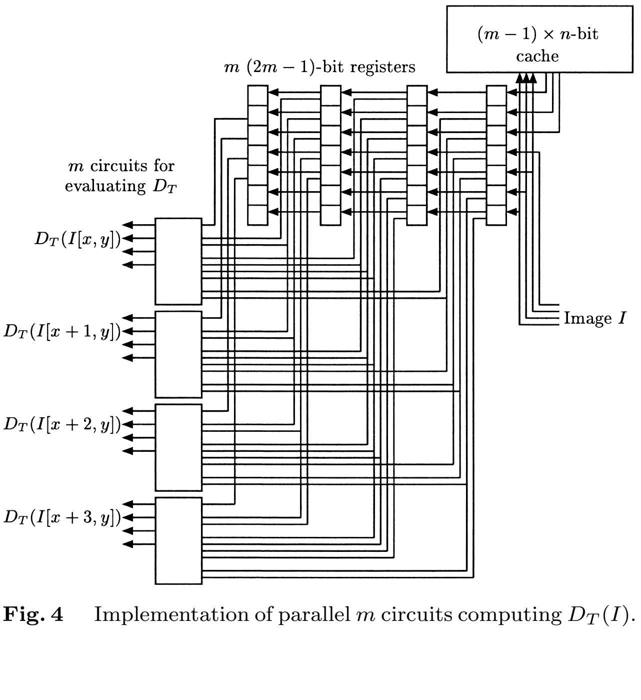

$(m-1)\times n$-bit cache
m $(2m-1)$-bit registers
m circuits for evaluating D_T
$D_T(I[x,y])$
$D_T(I[x+1,y])$
$D_T(I[x+2,y])$
$D_T(I[x+3,y])$
Image I

Fig. 4 Implementation of parallel m circuits computing $D_T(I)$.

In what follows, we will describe how our parallel image matching hardware illustrated in Fig. 4 works. An image I is transferred to the registers as follows.

```
for $i \leftarrow 1$ to $\frac{n}{m}$ do
  for $j \leftarrow 1$ to $n$ do
    Perform the following four operations in parallel:
    1. $I_m[(i-1)\cdot m+1, j]$ is transfered to the
       rightmost register;
    2. $I_{m-1}[(i-1)\cdot m+2, j]$ is transfered to the
       address $j$ of the cache;
    3. the $m-1$ pixels stored in the address $j$ of the
       cache is transferred to the rightmost register;
    4. the $m$ registers are shifted by one to the left.
```

In the first n clock cycles, $I_m[1,1], I_m[1,2], \ldots, I_m[1,n]$ are transferred in turn to the registers. Thus, in clock cycle j ($m \leqq j \leqq n$), the register is storing $I[1, j-m+1]$. Also, in these n clock cycles, the cache is storing $I_{m-1}[2,1], I_{m-1}[2,2], \ldots, I_{m-1}[2,n]$. In the following n clock cycles, $I_{m-1}[2,1], I_{m-1}[2,2], \ldots, I_{m-1}[2,n]$ are transferred from the cache to the registers. In the same time, $I_m[m+1,1], I_m[m+1,2], \ldots, I_m[m+1,n]$ are transfered to the registers through the PCI bus. Hence, in these n clock cycles, $I_{2m-1}[2,1], I_{2m-1}[2,2], \ldots, I_{2m-1}[2,n]$ are transferred to the registers in turn. Thus, in clock cycle $j+n$ ($m \leqq j \leqq n$), the registers are storing $I[2,j], I[3,j], \ldots, I[m+1,j]$ in turn. Again, in these n clock cycles, the cache stores $I_{m-1}[m+2,1], I_{m-1}[m+2,2], \ldots, I_{m-1}[m+2,n]$. Continuing similarly, every subimage $I[x,y]$ ($1 \leqq x,y \leqq n-m+1$) is stored in the register in some clock cycle.

As illustrated in Fig. 4, m circuits for evaluating D_T are connected to the m $(2m-1)$-bit registers. When the registers are storing $I[x,y], I[x+1,y], \ldots, I[x+m-1,y]$, the m circuits compute $D_T(I[x,y]), D_T(I[x+1,y]), \ldots, D_T(I[x+m-1,y])$ in parallel. Their minimum is computed by a tree of comparators in obvious way. Consequently, our parallel hardware computes $D_T(I)$ in $n\cdot\frac{n}{m} = \frac{n^2}{m}$ clock cycles. Further it has $O(m^2)$ flip-flops for registers, $O(me)$ gates, and an $(m-1)n$-bit block RAM.

5. Parallel Image Matching for Gray-Scale Images

This section is devoted to extend our image matching hardware for binary images to support gray-scale images. We start with a non-parallel image matching hardware using a single circuit computing $D_T(I[x,y])$. We then go on to show a parallel image matching hardware.

Each pixel of an L-level gray-scale image can be represented by $\log L$ bits. Thus, we use m m-digit registers with each digit storing $\log L$ bits. Figure 5 illustrates a part of the hardware implementation that computes $D_T(I)$ stored in the registers for $m=4$. Let

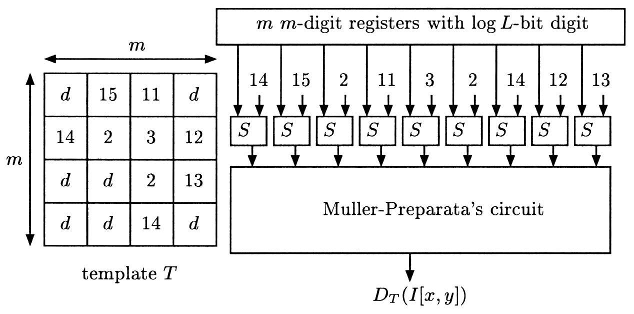


Fig. 5 Implementation of the image matcher for gray-scale images.

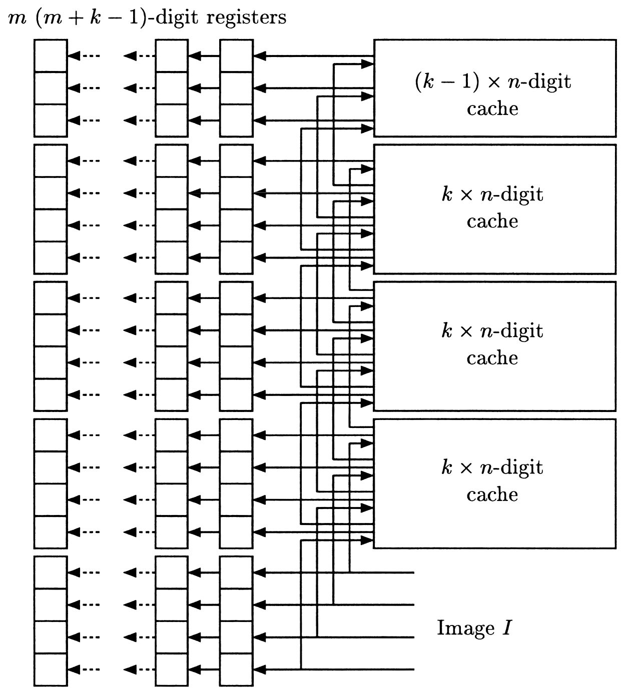


Fig. 6 Implementation of parallel image matcher for gray-scale images.

$I[x,y]$ denote an $m \times m$ pixel image currently stored in the registers. To compute $D_T(I[x,y])$, the difference for each non-"don't care" pixel values are computed. In other words, $|T_{i,j} - I_{i,j}|$ is computed for all effective pixel (i,j) such that $T_{i,j} \neq d$. In the figure, S denotes the circuit for computing this value. Note that each $T_{i,j}$ is a fixed value. Each S is a simple circuit that computes the absolute value of the sum of an $\log L$-bit integer and a constant integer. Thus, the circuit to compute the sum is simpler than the usual addition circuit. We use the Muller-Preparata's circuit to compute the sum of them, $\sum_{T_{i,j} \neq d} |T_{i,j} - I_{i,j}|$. Note that the Muller-Preparata's circuit computes the number of 1's in binary numbers. We can modify the circuit to compute the sum of integers. The Muller-Preparata's circuit has a recursive structure. It partitions the input into two groups, computes the sum within each group recursively, and the sum of the sums is computed by a ripple-carry adder. The readers should have no difficulty to confirm that the same technique can be applied to compute the sum of integers.

Next, we are going to show a parallel gray-scale image matching hardware using the image matcher illustrated in Fig. 5. Again, we assume that m bit of data can be supplied in every local clock cycle. Since each pixel is represented by $\log L$ bits, $\frac{m}{\log L}$ pixels in image I can be transferred in a local clock cycle. Let $k = \frac{m}{\log L}$. Our hardware uses k circuits that computes $D_T(I[x,y]), D_T(I[x+1,y]), \ldots, D_T(I[x+k-1,y])$ in parallel. Figure 6 illustrates the registers and the caches for $m = 16$ and $\log L = 4$. We use m registers of $(m+k-1)$ digits with each digit being $\log L$ bits, $\log L - 1$ caches of $k \times n$ digits, and a cache of $(k-1) \times n$ digits. We are going to show how the registers and the caches in Fig. 4 work. It transfers an image I trough the m-bit PCI bus as follows.

```
**for** $i \leftarrow 1$ **to** $\frac{n}{k}$ **do**
  **for** $j \leftarrow 1$ **to** $n$ **do**
    the following three operations are done in parallel:
    1. $I_k[(i-1) \cdot k + 1, j]$ is transfered to the rightmost
       register and to the address $j$ of the bottom cache;
    2. the $k$ pixels stored in the address $j$ of the cache
       is transferred to the rightmost register and to the
       cache above (if exists);
    3. the $m$ registers are shifted by one to the left.
```

In the first n clock cycles, the bottom cache is storing pixels in the first k rows of I. They are transferred to the cache above in the next n clock cycles. Since this process is repeated, the $(\log L - 1)$ caches from the bottom are storing the first $m-k$ rows of I in the first $n(\log L - 1)$ clock cycles. Hence, in the following n cycles, $I_m[1,1], I_m[1,2], \ldots I_m[1,n]$ are transferred to the register from the cache and from the PCI bus. Thus, the registers store $I[1,1], I[1,2], \ldots I[1, n-m+1]$ in turn in these n cycles. Next, in the following n cycles, $I_{m-1}[2,1], I_{m-1}[2,2], \ldots I_{m-1}[2,n]$ are transferred from the caches to the registers in turn. In the same time, $I_k[m+1,1], I_k[m+1,2], \ldots I_k[m+1,n]$ are transferred to the register though the PCI bus. Thus, the registers store $I[x,y]$ $(2 \leq x \leq k+1; 1 \leq y \leq n-m+1)$ in these n clock cycles. By the same process, every subimage $I[x,y]$ $(1 \leqq x, y \leqq n-m+1)$ is stored in the register in some clock cycle. Therefore, $D_T(I)$ is computed in $\frac{n^2}{k} = \frac{n^2 \log L}{m}$ clock cycles. Each Muller-Preparata's circuit uses $O(e \log L)$ gates. Thus, our hardware for gray-scale images uses $k \cdot O(e \log L) = O(me)$ gates. Since $\log L$ flip-flops are used for a digit in registers, our hardware uses $O(m^2 \log L)$ flip-flops for registers. It also uses $\log L$ block RAMs with no more than $kn \log L = mn$ bits for the caches.

6. The Performance Evaluation

The main purpose of this section is to evaluate the performance of our image matching hardware to compute $D_T(I)$ for templates T with 32×32 pixels and images I with 1024×1024 pixels.

Before we evaluate the performance of our hardware, we will show the computing time by the software approach as counterparts. Table 1 shows the computing time of $D(T, I)$ on a 2.4 GHz Pentium4-based Windows 2000 PC. The software is written in C-language and complied using Visual C++ (Ver. 6.0). For $L = 4$ and 16, the value of $D(T, I)$ is computed by evaluating formulas (1) and (3) in Sect. 2 combined. Since $D(T, I[x, y])$ is evaluated using formula (1), the computing time is proportional to e. For $L = 2$, we use the bitwise XOR operation of a word of 32-bit data to evaluate formula (2). Also, we accelerate the computation of the sum in formula (2) using the look-up table storing the number of 1's in a 16-bit data. More precisely, let N be a table of size $5 \times 2^{16} = 320$ Kbits such that $N[x]$ is storing the number of 1's in a 16-bit word x. The number of 1's in a word of 32-bit data can be computed by looking up table N twice. Note that the computing time in Table 1 does not include the time necessary to build the table N.

We have tested our image matching hardware using Spartan2 (XC2S150) PCI card Strathnuey [6]. This PCI card is connected to the host PC through the 33-MHz 32-bit PCI bus. We complied the generated Verilog HDL source using Xilinx ISE Foundation (Ver. 4.2). Table 2 illustrates the performance of our image matching hardware which includes the clock frequency given by the timing analyzer, the actual time to evaluate $D_T(I)$, the speed-up over the software, the number of used slices over 1728 available slices, and the number of used slice flip-flops over 3456 available flip-flops. Unfortunately, due to the small capacity of XC2S150, we could test our non-parallel hardware for binary images with $e = 128, 256, 512, 768$ effective pixels and 4-level gray-scale images with $e = 128, 256$. For our parallel image matching hardware, we could test for binary images with only $e = 16$ pixels. Consequently, even if we use non-parallel image matching hardware embedded in the small FPGA, $D_T(I)$ can be computed more than 16 times faster than the software. The computing time for binary images is approximately 75 msec, which is bounded by the bandwidth of the 33 MHz 32-bit PCI bus. More precisely, since $1024 \times (1024 - 32 + 1) = 1.016$M words of 32-bit data are transfered in 75 msec, the PCI bus sends images in 434 Mbit/s, which is close to the actual maximum ability of the 33 MHz 32-bit PCI bus. If the 66 MHz 64-bit PCI bus is available, we can expect that $D_T(I)$ is computed faster.

We have also estimated the performance of our image matching hardware using the VirtexII series FPGA XC2V8000. Again, we assume that a template image T has 32×32 pixels and an image I has 1024×1024 pixels. We have estimated our hardware for randomly generated templates T of size 32×32 with effective pixels $e = 128, 256, 512$ and 768. Table 3 shows the performance of our non-parallel image matching hardware which uses a single circuit for computing $D_T(I)$. The table includes the clock frequency (MHz), the estimated computing time of $D_T(I)$ (msec), the speedup factor over the software solution in Table 1, the number of used slices out of 46592 available slices, and the number of used slice flip-flops out of 93184 available slice flip-flops. We assume that the PCI bus has enough bandwidth. More precisely, we assume that, if a hardware runs in local clock with frequency 56.4 MHz, the PCI bus can send pixels in at least 32 bit $\times$ 56.4 MHz $=$ 1804.8 Mbit/s. Since 64-bit 66-MHz PCI bus is currently available [9], this assumption is reasonable. For $L = 4$ and 16, our hardware uses 2, and 6 18 K-bit blocks out of 168 block RAMs, respectively. These block RAMs are used for the image cache. For binary image (i.e. $L = 2$), it uses no block RAM. Even if we use only a single circuit computing $D_T(I)$, the speedup factors over the software solution are in the range 67-333.

Table 4 shows the performance of parallel image matching hardware. For $L = 2, 4$, and 16, we use

Table 1 The computing time of $D(T, I)$ by software (msec).

$e =$	128	256	512	768
$L = 2$	1256	1345	1652	1797
$L = 4$	1495	2958	5812	8667
$L = 16$	1530	3042	5970	8918

Table 2 The performance of our image matching hardware implemented on Spartan2.

L	e	Freq. (MHz)	Time (msec)	Speed -up	Slices	Slice FFs
2	128	39.6	75.2	16.7	714	902
2	256	35.8	75.2	17.9	926	1018
2	512	32.7	75.0	22.0	1316	1107
2	768	30.1	75.1	23.9	1681	1122
4	128	33.0	76.5	19.5	1334	1747
4	256	34.3	76.2	38.7	1726	1978
2	16	24.1	4.9	-	1682	1741

Table 3 The performance of our image matching hardware implemented on VirtexII.

L	e	Freq. (MHz)	Time (msec)	Speed -up	Slices	Slice FFs
2	128	56.4	18.0	69.8	708	882
2	256	51.0	19.9	67.6	927	1029
2	512	46.5	21.9	75.4	1265	1100
2	768	42.7	23.8	75.5	1594	1127
4	128	51.5	20.1	74.4	1295	1755
4	256	46.8	22.1	134	1713	2027
4	512	43.0	24.0	242	2368	2145
4	768	39.7	26.0	333	2992	2193
16	128	45.5	22.9	66.8	2661	3412
16	256	41.9	24.9	122	3680	3953
16	512	38.7	26.9	222	5341	4188
16	768	36.0	28.9	309	7282	4284

Table 4 The performance of our parallel image matching hardware implemented on VirtexII.

L	e	Freq. (MHz)	Time (msec)	Speed -up	Slices	Slice FFs
2	128	28.0	1.17	1074	6188	2277
2	256	26.2	1.25	1076	11307	2899
2	512	24.6	1.33	1242	21765	4557
2	768	23.1	1.42	1265	32310	6243
4	128	29.3	2.24	667	6457	3197
4	256	27.3	2.40	1233	11403	3663
4	512	25.6	2.57	2261	21659	5276
4	768	24.2	2.71	3198	32056	7046
16	128	29.9	4.39	349	8860	5071
16	256	27.8	4.72	644	15021	5184
16	512	26.1	5.03	1187	27146	5264
16	768	24.6	5.31	1679	42251	5433

Table 5 The performance of non-instance-specific image matching hardware implemented on VirtexII.

L	Freq. (MHz)	Time (msec)	Slices	Slice FFs
2	34.9	29.14	3545	3154
4	32.5	31.79	6395	5256
16	27.0	38.57	13865	9418

$\frac{m}{\log L} = 32, 16$, and 8 circuits that compute $D_T(I)$ in parallel, respectively. For $L = 16$ and $e = 768$, the hardware uses $\frac{42251}{46592} \approx 90.7\%$ of available slices. For $L = 2, 4$ and 16, the hardware also uses 2, 4, and 8 block RAMs out of 168 block RAMs. They are used for the image cache illustrated in Figs. 4 and 6. The speedup factors over the software solution are in the range 349-3198.

As counterparts, we have also evaluated the performance of non-instance-specific solution for the image matching hardware. More specifically, the value of each pixel in template T is stored in a flip-flop in the FPGA. In this case, template T can be changed without generating and compiling the Verilog HDL source. The non-instance-specific hardware for an $m \times m$ L-level gray scale template image T uses additional $m^2 \log L$ flip-flops for the value of pixels in the range $[0, L-1]$. Table 5 shows the performance of the hardware using the non-instance-specific approach. The computing time is more than 20% longer than that of our instance-specific solution shown in Table 3. Also, the non-instance-specific hardware uses approximately twice as many as slices than the instance-specific hardware for $e = 768$.

We have also tried to embed the parallel version of the non-instance-specific hardware in the FPGA. However, the compilation of the Verilog HDL source could not be completed due to insufficient slices in XC2V8000. Thus, the instance-specific solution shown in this paper is much more efficient than the conventional non-instance-specific solution if the pre-computation time for generating and compiling the Verilog HDL source is allowed.

7. Conclusions

We have presented an FPGA-based instance-specific solution for an image retrieve problem, that asks to lists all images in a database having similar subimage to a particular template image T. Our hardware for L-level gray-scale images runs in $\frac{n^2 \log L}{m}$ local clock cycles. Further, the timing analyzer shows that this hardware attains up to 3000 speed-up factor over the software solution.

Acknowledgments

This work is supported in part by the Ministry of Education, Science, Sports, and Culture, Government of Japan, Grant-in-Aid for Exploratory Research (14658090). The authors would like to thank to Professor Tetsuo Asano for helpful comments, and anonymous referees who give detailed comments to improve the presentation of this paper.

References

[1] J.L. Bordim, Y. Ito, and K. Nakano, "Accelerating the CKY parsing using fpgas," Proc. International Conference on High Performance Computing, pp.41–51, 2002.

[2] Y. Futamura, K. Nogi, and A. Takano, "Essence of generalized partial computation," Theoretical Computer Science, vol.90, no.61–79, 1991.

[3] M. Gavrilov, P. Indyk, R. Motwani, and S. Venkatasubramanian, "Geometric pattern matching: A performance study," Proc. Symposium on Computational Geometry, pp.79–85, 1999.

[4] N.D. Jones, C.K. Gomard, and P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[5] T. Kean and A. Duncan, "A 800 Mpixel/sec reconfigurable image correlator on XC6216," Proc. International Conference on Field Programmable Logic and Applications (FPL), pp.382–391, 1997.

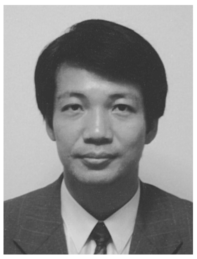
[6] Nallatech Ltd., Strathnuey SPATAN-II PCI card users guide, 2000.

[7] D.E. Muller and F.P. Preparata, "Bounds to complexityies of network for sorting and for switching," J. ACM, vol.22, no.195–201, 1975.

[8] K. Nakano and K. Wada, "Integer summing algorithms on reconfigurable meshes," Theoretical Computer Science, vol.197, no.57–77, 1998.

[9] T. Shanley and D. Anderson, PCI System Architecture, 4th ed., Addison Wesley, 1999.

[10] M.J. Wirthlin and B. McMurtrey, "Efficient constant coefficient multiplication using advanced fpga architecture," Proc. International Conference on Field Programmable Logic and Applications (FPL), pp.555–564, 2001.



Koji Nakano received the BE, ME and Ph.D degrees from Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992–1995, he was a Research Scientist at Advanced Research Laboratory, Hitachi Ltd. He had worked at the Department of Electrical and Computer Engineering, Nagoya Institute of Technology until 2001. He is currently an Associate Professor with the School of Information Science, Japan Advanced Institute of Science and Technology. He has published papers in IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Computing, Theoretical Computer Science, Theory of Computing Systems, Parallel Algorithms and Applications, and IEICE Transactions. He has served as an editor of Journal of Information Processing Society of Japan, IEICE Transactions on Information and Systems, International Journal of Foundations on Computer Science. He has served on the program chair and committee member of conferences and workshops including International Conference on Parallel Processing, Workshop on Advances in Parallel and Distributed Computational Models, Workshop on Wireless Networks and Mobile Computing, and Reconfigurable Architecture Workshop. His research interests includes mobile computing, hardware algorithms, reconfigurable computing, parallel algorithms and architectures, computational complexity, and graph theory.

Etsuko Takamichi received the BE degree from Niigata University, Japan in 2001. She is currently a master student in Japan Advanced Institute of Science and Technology.