Task Allocation with Algorithm Transformation for Reducing Data-Transfer Bottlenecks in Heterogeneous Multi-Core Processors: A Case Study of HOG Descriptor Computation

Hasitha Muthumala WADYASOORIYA (a), Student Member, Daisuke OKUMURA†, Nonmember, Masanori HARIYAMA†, Member, and Michitaka KAMEYAMA†, Fellow

SUMMARY Heterogeneous multi-core processors are attracted by the media processing applications due to their capability of drawing strengths of different cores to improve the overall performance. However, the data transfer bottlenecks and limitations in the task allocation due to the accelerator-incompatible operations prevents us from gaining full potential of the heterogeneous multi-core processors. This paper presents a task allocation method based on algorithm transformation to increase the freedom of task allocation. We use approximation methods such as CORDIC algorithms to map the accelerator-incompatible operations to accelerator cores. According to the experimental results using HOG descriptor computation, the proposed task allocation method reduces the data transfer time by more than 82% and the total processing time by more than 79% compared to the conventional task allocation method.

key words: heterogeneous multi-core processor, task-allocation, system-on-chip

1. Introduction

Heterogeneous multi-core processors provide a platform to implement complicated media processing applications power-efficiently. Heterogeneous multi-core processors have different processor cores integrated on the same chip. If the tasks of an application are correctly allocated to the most suitable processor cores, all the cores work together to increase the overall performances. Examples of heterogeneous multi-core processors are [1] and [2]. The former has multiple cores of CPUs and ALU arrays. The latter has multiple cores of CPUs, a micro-controller and SIMD (single-instruction multiple-data) type multi-bank matrix processors (MBMX).

In heterogeneous multi-core processors, different tasks are allocated to different processor cores. Therefore, processor cores need to access each other’s data through the inter-connection network. The accelerator cores contain several local memories for parallel data access. Therefore, whenever a data transfer with an accelerator core occurs, we need to transfer data from the global memory to the accelerator core’s local memories. As a result, a large amount of time is wasted for data transfers and that increases the total processing time. The data transfer time is a huge problem especially in media processing applications that involve a huge amount of pixel data.

To solve this problem, we need to allocate the successive tasks with data dependencies to the same processor core, so that one task’s output is immediately used by the other tasks without transferring it to the global memory. In such situations, it is better to assign both tasks to the accelerator cores, since it gives a higher processing speed. However, freedom of assigning tasks to accelerator cores has limitations. In heterogeneous multi-core processors, the accelerator cores are usually designed to operate at low-power by limiting their calculation capabilities to simple operations such as additions, multiplications, bit-shifting, etc. We call these, accelerator-compatible operations. However, accelerator cores such as FE-GA (Flexible Engine/General ALU Array) [1] and MBMX [2] cannot perform complex operations such as square-root calculation and trigonometric operations. We call these complex operations, accelerator-incompatible operations. Therefore, if a task contains accelerator-incompatible operations, that task cannot be assigned to the accelerator cores. As a result, full potential of the heterogeneous multi-core processors cannot be achieved.

In this paper, we show that the algorithm transformation can increase the freedom of task allocation effectively. We use the task allocation of the HOG descriptor computation as a case study and show that the processing time can be reduced substantially.

2. Previous Works

The application implementation on heterogeneous multi-core processors has been done in many previous works [3]–[5]. However, in these works, tasks with data-parallel and accelerator-compatible operations are assigned to the accelerator cores and the other tasks are assigned to the CPU cores. We call this task allocation as “conventional task allocation.” In the conventional task allocation, data-transfer bottlenecks are a big problem, since they increase the processing time significantly.

Previous works such as [6] and [7] discussed the task allocation algorithms considering communication (data transfer) overhead. Communication-Aware Allocation and Scheduling Framework is proposed in [6]. Its hardware model contains homogeneous ARM (Advanced RISC Ma-
chine) processors interconnected by an AMBA (Advanced Microcontroller Bus Architecture) bus. Targeted applications are stream-oriented pipelined-tasks. It solves the task mapping in a two-step process: task allocation followed by task scheduling. The task allocation problem is solved through the integer-programming while the tasks scheduling problem is solved through the constraint-programming. A heuristic algorithm to map tasks dynamically to heterogeneous MPSoC (Multiprocessor Systems-on-Chip) is discussed in [7]. It considers a cost function that depends on the communication volume (data transfer volume) to map tasks. It starts with an initial task-map and allocates new tasks dynamically using a heuristic algorithm that considers the communication cost and the type of the task.

However, none of the previous works consider algorithm transformation to remove data transfer bottlenecks and the maximum speed-up of the heterogeneous multi-core processors is not observed. This work considers algorithm transformation to increase the freedom of task allocation effectively and to remove data transfer bottlenecks. We use the task allocation of the HOG descriptor computation as a case study and show that the processing time can be reduced substantially.

3. Task Allocation Considering the Data Transfer Time

In the conventional task allocation method, data-parallel and accelerator-compatible tasks are assigned to the accelerator cores and the other tasks are assigned to the CPU cores. Such an example is shown in Fig. 1. In this example, the tasks 1, 2 and 3 contain data-parallel operations. However, the task 2 is allocated to the CPU core since it contains accelerator-incompatible operations. As a result, the data transfer time is very large between the tasks 1 and 2. However, if we can assign the task 2 to the accelerator core, we can reduce this data transfer time. In this paper, we consider algorithm transformation of task 2 using approximation methods such as CORDIC (coordinate rotation digital computer) algorithms [8]. Therefore, we can increase the freedom of task allocation and assign the task 2 to the accelerator core to decrease the processing time as shown in Fig. 2.

CORDIC is a simple and efficient algorithm to calculate complex functions such as hyperbolic and trigonometric functions using only additions, subtractions and bit-shift operations. Since most accelerator cores are capable of these operations, we can easily use CORDICs for algorithm transformation. Other than CORDIC, there are many approximation methods such as power series, table-lookups, etc.

We can change accelerator-incompatible operations into accelerator-compatible operations by using very simple methods. For example, division can be changed into multiplication by multiplying both sides by the denominator as shown in Eq. (1).

\[ f = \frac{s}{y} \rightarrow f \times y = s \]  

Figure 3 shows the flow-chart of the division. The calculation is done in \( N \) iterations using a series of addition, multiplication, bit-shift and comparison operations. Figure 4 shows an example of the division. In this example, we calculate the division of 267 by 3 in 8 iterations using 128 as
the Base value. We calculate \((f + \text{Base}) \times 3\) value and compare it with 267. In each step, Base value reduces by half by shifting one bit to the right. If \((f + \text{Base}) \times 3 \leq 267\), we add the Base value to \(f\). After 8 iterations, we get the division result. The circuit implementation is very similar to that of square-root operation in Sect. 4.2.1.

The square-root operation can be simplified into multiplication by taking the product with itself as shown in Eq. (2).

\[
f = \sqrt{y} \rightarrow f^2 = y
\]  

Figure 5 shows the flow-chart of the square-root calculation. The calculation is done in \(N\) iterations using a series of addition, multiplication, bit-shift and comparison operations. Figure 6 shows an example of square-root calculation. In this example, we calculate the square-root of 676 using 128 as the Base value. We calculate \((f + \text{Base}) \times (f + \text{Base})\) value and compare it with 676. In each step, Base value reduces by half by shifting one bit to the right. If \((f + \text{Base}) \times (f + \text{Base}) \leq 676\), we add the Base value to \(f\). After 8 iterations, we get the square-root. The circuit implementation on the accelerator core is discussed in Sect. 4.2.1. Note that, the division and square-root algorithms discussed are based on CORDIC but contains multiplications. Therefore, these algorithms can be implemented in an accelerator core that contains multipliers.

We make a library that contains these algorithm transformation methods for many complex operations. When we have a task with accelerator-incompatible operations, we replace such operations with an accelerator-compatible algorithm from the library. Then we perform an exhaustive search to find the best task allocation.

To evaluate the effectiveness of CORDIC algorithms in accelerator cores, we compare the processing time of some complex operations on CPU and accelerator cores. We use the accelerator core called FE-GA [1]. A detailed description on the functionality of the FE-GA is given in Sect. 5. Figure 7 shows the processing time for square-root calculation using the CPU core and the accelerator core with CORDICs. According to the results, the CPU core is faster when the data amount is small. When the amount of data increases to a particular cross-point, the processing time of the CPU core exceeds that of the accelerator core. In this example, the cross-point is 18 square-root calculations. For a smaller amount of data, the control overhead of the accelerator core is larger than the data processing time. The control overhead refers to the initialization time of an accelerator core. When the data amount is larger, the control overhead is negligible compared to the data processing time. Moreover,
accelerator employs parallel processing to decrease the processing time. In this example, we can see that the processing time of the accelerator core is many times smaller than that of the CPU core for a large amount of data. Therefore, the algorithm transformation for accelerator-incompatible tasks is very effective.

However, considering only the processing time does not give the best results for heterogeneous processors. Figure 8 shows the processing time comparison for divisions in CPU and accelerator cores. In this case, we assume that the task after the division is assigned to the accelerator core. Therefore, we have to transfer data from the CPU core to accelerator core. After considering the data transfer time, the cross-point has shifted from \( C_g \) to \( C_c \) as shown in Fig. 8. Therefore, not only the processing time, but also the data transfer time has to be considered in the task allocation.

To determine which task allocation is the best, we need to know the total processing time. The total processing time determined by 3 factors: processing time in accelerators, processing time in CPUs and the processing time for the data transfers. The processing time of the accelerator cores mainly depends on the amount of data, the parallelism and the clock frequency. Usually, accelerators are used for parallel-data processing and they do not contain data dependent control paths. Therefore, we use Eq. (3) to estimate the processing time \( T_a \) in an accelerator core.

\[
T_a = \frac{N_{data}}{P \times f} + C
\]  

(3)

The amount of data is \( N_{data} \), the number of parallel calculations is \( P \), the frequency is \( f \) and the control overhead is \( C \). The control overhead refers to the initialization time of the accelerator core. To execute a task in an accelerator core, we need to set its control registers and also need to download configuration data from the configuration memory to the registers. The time required for these are called the initialization time. Equation (3) is difficult to apply when the number of loop iterations is small. However, it can be applied successfully in media processing applications that usually process a large amount of data. When we have loop-carried dependencies in tasks, the amount of parallel calculations decreases. That is, \( P \) in Eq. (3) decreases. Therefore, we have to adjust \( P \). In the worst case, every iteration in a loop depends on their earlier iterations. As a result, we have to run the loop in serial, so that \( P \) equals to 1.

The processing time for CPUs is very difficult to estimate due to the complex control processing and data dependent control paths. Therefore, we implement each task on CPUs in advance. Then we measure the processing time and use it for the optimization process. Since CPU cores are easily programmable using “C language”, the actual implementation is not difficult.

The processing time consumed by transferring one core’s (source-core’s) data to another core (destination-core) is called the data transfer time. The data transfer time depends on the data amount and the data transfer rate. The data amount must be equal to the number of output data of the source-core. However, in heterogeneous multi-core processors, the output data may have to copy more than once to the local memories of the destination-core. An effective memory allocation can solve this problem. Previous works [9] have discussed this issue in detail. However, memory allocation is a time consuming problem. The memory allocation can be different for each different task allocation results. Since, we use an exhaustive search in this paper, the number of task allocation results are very large. As a result, the memory allocation time is also very large. Therefore, in the processing time estimation, we do not perform the memory allocation. We assume that the memory allocation is done perfectly, so that the output data are copied only once to the local memories. Under this assumption, we use Eq. (4) to estimate the data transfer time. The data transfer time is \( T_d \) and the data transfer rate is \( r \).

\[
T_d = N_{data} \times r
\]  

(4)

The total processing time equals to the sum of all the data transfer times, the processing time in CPU cores and the processing time in the accelerator cores.

To evaluate the accuracy of the estimation, we compare the estimated results with the measured results obtained after the implementation. The accuracy is calculated using Eq. (5). Table 1 shows the estimated and measured processing time. According to the results, the estimation has a sufficient accuracy of over 95%.

\[
\text{Accuracy} = \left(1 - \frac{|\text{Measured} - \text{Estimated}|}{\text{Measured}}\right) \times 100\%
\]  

(5)

Figure 9 shows the task allocation process. At the beginning, we divide a given algorithm into several tasks. Note that, the task partitioning is done manually and automation of task partitioning is not in the scope of this paper. After the task allocation, we get two set of tasks where one

<table>
<thead>
<tr>
<th>Table 1 Processing time estimation.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>SAD calculation in accelerators</td>
</tr>
<tr>
<td>Estimated</td>
</tr>
<tr>
<td>6.16</td>
</tr>
<tr>
<td>Data transfer</td>
</tr>
<tr>
<td>12.49</td>
</tr>
</tbody>
</table>
set is accelerator compatible and the other set is accelerator incompatible. Then we do the algorithm transformation for accelerator incompatible tasks by choosing an algorithm from the library. This will give us a set of tasks that are compatible with accelerator cores. Then we calculate the processing time of each task in both CPU and accelerator. To calculate the processing time in CPU, we run the particular task in CPU and measure the processing time. To estimate the processing time in an accelerator core, we use Eq. (3). Then we consider all possible task allocation results and evaluate for total processing time. When calculating the total processing time, we consider the data transfer time among tasks. If two data dependent tasks are mapped to two different cores, we use Eq. (4) to estimate the data transfer time. We perform an exhaustive search to find the best task allocation with minimum processing time.

4. A Case Study of HOG Descriptor Computation

4.1 Algorithm of HOG Descriptor Computation

We use HOG (histogram of oriented gradients) algorithm [10] to evaluate the proposed task allocation method. The HOG descriptor is widely used in computer vision and image processing for object detection [11], [12]. It contains four major steps.

**Step 1: Gradient computation**

In the step 1, 1-D derivative mask is applied to all the pixels in the image in horizontal and vertical directions as shown in Fig. 10. Equations (6) and (7) gives the horizontal and vertical derivatives respectively. The color or the intensity of the pixel at the coordinates \(x, y\) is given by \(I(x, y)\).

\[
\begin{align*}
    f_x(x, y) &= I(x + 1, y) - I(x - 1, y) \\
    f_y(x, y) &= I(x, y + 1) - I(x, y - 1)
\end{align*}
\]  

**Step 2: Orientation binning**

An image is divided into “cells” where each cell contains a group of pixels as shown in Fig. 11. Note that the cells do not overlap each other. The second step of HOG descriptor computation involves creating histograms for each cells. Each pixel within a cell casts a weighted vote for an orientation-based histogram-channel using the values found in the gradient computation. A histogram-channel or a “bin” refers to a set of orientations (angles) between two values. Figure 12 shows the creation of a histogram and the bins. The orientation is given by Eq. (8).

\[
    \theta(x, y) = \tan^{-1} \left( \frac{f_y(x, y)}{f_x(x, y)} \right)
\]  

The bins are evenly spread over \(-90^\circ\) to \(90^\circ\). Equation 9 is
used to find the bins of the orientations.

\[
\alpha_n \leq \theta(x, y) < \alpha_{n+1}
\]

such that \( \alpha_n \in \{-90^\circ, \ldots, 90^\circ\} \) (9)

As for the vote weight, we take the square-root of the gradient magnitude as given in Eq. (10).

\[
m(x, y) = \sqrt{f_x(x, y)^2 + f_y(x, y)^2}
\] (10)

**Step 3: Descriptor blocks**

A group of neighboring cells is called a block as shown in Fig. 11. All the cells in an image are grouped into blocks such that the blocks can be overlapped each other.

**Step 4: Block normalization**

The HOG descriptor is the vector of the components of the normalized cell histograms of a block. If \( V_k \) is the vector of a block and \( \epsilon \) is a very small value close to zero, the HOG descriptor \( V \) is given by Eq. (11).

\[
V = \frac{V_k}{\sqrt{||V_k||^2 + \epsilon}}
\] (11)

### 4.2 Task Partitioning and Algorithm Transformation

We partition the HOG descriptor computation into 7 different tasks. In this work, the task partitioning is done manually. In the process of mapping applications to heterogeneous processors, we need a method to partition the tasks automatically. However, automation of task partitioning is one of our future works and it is not in the scope of this paper. This paper focuses on the optimal allocation of the partitioned tasks. The manually partitioned tasks are summarized here.

**Task 1:** Gradient computation on \( x \) direction.

\[
f_x(x, y) = I(x + 1, y) - I(x - 1, y)
\]

**Task 2:** Gradient computation on \( y \) direction.

\[
f_y(x, y) = I(x, y + 1) - I(x, y - 1)
\]

**Task 3:** Gradient magnitude computation.

\[
M(x, y) = f_x(x, y)^2 + f_y(x, y)^2
\]

**Task 4:** Square-root of the gradient magnitude.

\[
m(x, y) = \sqrt{M(x, y)}
\]

**Task 5:** Orientation computation

\[
\theta(x, y) = \tan^{-1} \frac{f_y(x, y)}{f_x(x, y)}
\]

\[
\alpha_n \leq \theta(x, y) < \alpha_{n+1}
\]

**Task 6:** Histogram creation.

\[
\text{Bin}_{\alpha_n} = \sum m
\]

**Task 7:** Block assignment.

**Task 8:** Block normalization.

\[
V = \frac{V_k}{\sqrt{||V_k||^2 + \epsilon}}
\]

According to the task partitioning of the HOG algorithm, there are 8 tasks. The tasks 1, 2 and 3 contain subtraction, multiplication and addition operations which are accelerator-compatible. The task 4 contains square-root calculation which is accelerator incompatible. The task 5 contains division, arctangent and comparison operation. The division and the arctangent are accelerator-incompatible while the comparison is accelerator-compatible. Task 6 contains addition which is accelerator-compatible. The task 8 contains division, square-root, multiplication and addition. Division and square-root is accelerator-incompatible while addition and multiplication are accelerator-compatible. Therefore, in this task partition, we have 3 acceleration-incompatible complex operations: division, square-root and arctangent. Therefore, we cannot assign these 3 tasks to the accelerator cores without algorithm transformations. The task 7 and 8 have calculations that need data paths with a large bit-width. However, the accelerator we used have only 16-bit data paths. Therefore, we only consider the task allocation of tasks 1 to 6. The assignment of the task 7 and 8 are fixed to the CPU core.

We used a heterogeneous multi-core processor called RP1 [1] to implement the HOG descriptor computation. It has 4 homogeneous CPUs and 2 FE-GA cores as accelerators. Figure 13 shows the overall architecture of the processor. The data transfer among processor cores are done through the “SuperHyway” bus. An off-chip SDRAM of 128 MB is attached to the SuperHyway to store a large amount of image data that are received from image devices such as cameras. The processing speeds of FE-GAs and CPUs are 300 MHz and 600 MHz respectively. The architecture of the FE-GA core is shown in Fig. 14. It has an array of 32 16-bit processing elements (PEs). These PEs are dynamically reconfigured as adders, subtracers, logic gates etc. A PE can be connected to its four neighbors and those connections are also dynamically reconfigurable. The FE-GA has 10 local memories of 4 kB each which are called CRAMs (compiled random access memories). Note that, the operations such as square-root, division, trigonometric calculations and floating-point calculations are not available in the FE-GA cores.
4.2.1 Algorithm Transformation for Task 4

The task 4 contains square-root calculation which is accelerator-incompatible. The Square-root calculation can be implemented using power series, table-lookups or CORDIC algorithms. Power series contains many multiplications and requires many multipliers for the implementation. The accelerator cores such as FE-GAs with a small number of multipliers take many clock cycles to execute a power series. Table-lookups need a lot of memory. The FE-GA accelerator core has only 40 KBytes of local memory and most of it is used for the data storage. Therefore, it is difficult to use table-lookups. However, CORDIC algorithms, which has many additions and bit-shift operations, are relatively easier to implement in FE-GA accelerator cores with limited number of multipliers and local memories. In the implementation, we used the CORDIC-based algorithm discussed in Sect. 3. To implement it, we use the relationship in Eq. (12).

\[ m^2(x, y) = M(x, y) \tag{12} \]

Figure 15 shows the main operations of the implementation of square-root in FE-GA using 16 iterations. Since FE-GA is dynamically reconfigurable, we use 16 contexts. We use 1 shifter, 1 adder, 1 multiplier, 1 comparator and 1 multiplexer for a one data calculation. Since there are 32 PEs in FE-GA, we run 4 parallel operations. In each context, we perform 4096 operations in pipe-line. The output of one context is the input of another context. For example, the output \( m_1 \) of context 1 in Fig. 15 is the input of context 2. After executing all 16 contexts, we get the square-root values.

4.2.2 Algorithm Transformation for Task 5

In the task 5, we calculate the orientation (angle) of the gradient in each pixel using Eq. (8). Then we find the bins for all the orientations using Eq. (9). Figure 16 shows the relationship between the angle and its tangent. Between the
angles $-90$ to $90$, tan $\theta$ is a one-to-one function, where every value of \( \theta \) corresponds to only one value of tan $\theta$. Therefore, we use the tangent instead of the orientation as shown in Eq. (13).

$$\tan \theta = \frac{f_y}{f_x} \quad (13)$$

In the HOG algorithm, we have 9 bins from $-90^\circ$ to $90^\circ$ as shown in Fig. 12. To find the bins of the orientations, we use Eq. (14).

$$\tan \alpha_n \leq \tan \theta = \frac{f_y}{f_x} < \tan \alpha_{n+1} \quad (14)$$

$$\alpha_0 = -90^\circ, \alpha_1 = -70^\circ, \ldots, \alpha_9 = 90^\circ \quad n \in \{0, \ldots, 8\}$$

However, Eq. (14) contains division which is an accelerator-incompatible operation. Therefore, we change the division into multiplication by multiplying $f_x$ as shown in Eqs. (15) and (16). Note that, Eq. (16) is used when the $f_x$ is less than 0.

$$f_x \times \tan \alpha_n \leq f_y < f_x \times \tan \alpha_{n+1} \quad f_x > 0 \quad (15)$$

$$f_x \times \tan \alpha_n > f_y \geq f_x \times \tan \alpha_{n+1} \quad f_x < 0 \quad (16)$$

Since $\alpha_0$ to $\alpha_9$ are constants, the values of tan $\alpha_n$ are also constants. Therefore, we transformed task 5 into an algorithm that does not contain divisions and square-root operations.

Figure 17 shows the main operations of the implementation of task 5 in FE-GA. We use 2 multipliers, 2 comparators and 1 logic operator. In FE-GA, we can execute two data in parallel.

### Table 2 Library of accelerator-compatible operations for FE-GA.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of contexts</th>
<th>Parallelism</th>
<th>Number of clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square-root using CORDIC</td>
<td>16</td>
<td>4</td>
<td>$(15 + N_{data}/4) \times 16$</td>
</tr>
<tr>
<td>Square-root using power series</td>
<td>6</td>
<td>1</td>
<td>$(15 + N_{data}) \times 6$</td>
</tr>
<tr>
<td>Division</td>
<td>16</td>
<td>4</td>
<td>$(15 + N_{data}/4) \times 16$</td>
</tr>
<tr>
<td>Division using power series</td>
<td>6</td>
<td>1</td>
<td>$(15 + N_{data}) \times 6$</td>
</tr>
<tr>
<td>Arctangent using CORDIC</td>
<td>16</td>
<td>4</td>
<td>$(15 + N_{data}/4) \times 16$</td>
</tr>
</tbody>
</table>

### Table 3 Specifications of HOG implementation.

<table>
<thead>
<tr>
<th>Image size</th>
<th>$64 \times 64$ pixels (1 block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image type</td>
<td>8-bit grayscale</td>
</tr>
<tr>
<td>Block size</td>
<td>$8 \times 8$ cells</td>
</tr>
<tr>
<td>Cell size</td>
<td>$8 \times 8$ pixels</td>
</tr>
<tr>
<td>Number of bins</td>
<td>9 ($-90$ to $90$ degrees)</td>
</tr>
</tbody>
</table>

very small. In the implementation, we divide the image into 8 arrays of size $64 \times 8$. Then process the arrays one-by-one in the FE-GA. Table 4 summarize the tasks implementation on FE-GA accelerator core. It shows the number of contexts in each task, PEs and CRAMs per context, amount of parallelism and data amount (or loop iterations) in each task. Note that, the loop iterations given in Table 4 are executed in parallel. For example, in task 1, 4096 loop iterations are executed with a parallelism of 8. Therefore, the total number of control steps for task 1 is 4096/8.

Table 5 shows the allocation of tasks in conventional and proposed methods and their respective processing time. In the conventional method, the tasks 1, 2 and 3 are allocated to the FE-GA cores and the other tasks are allocated to the CPU cores. In this case, the data transfer time occupies 60% of the total processing time. That is, the data transfer time exceeds the calculation time. However, this is the best solution possible without the algorithm transformation. Using the algorithm transformation in the proposed method, we allocate the tasks 4, 5 and 6 to the FE-GA cores. As a result, the data transfer time reduced from 963 ms to 145 ms by more than 84%. The calculation time reduced from 663 ms to 185 ms by more than 72%. The total processing time reduced by more than 79%.

Figure 18 shows the comparison of the processing speeds of CPU-only implementation, the conventional task allocation and the proposed task allocation. According to the results, the conventional task allocation gives only 3.4 times of speed-up compared to the CPU-only implementation. This speed-up is no way close to the full potential of the heterogeneous multi-core processor. Using algorithm transformation in the proposed method, we increase the freedom of task allocation. As a result, we obtained a much better speed-up of 16.8 times compared to the CPU-only implementation. Therefore, the speed-up increased by more than 4.9 times compared to the conventional task allocation.

Table 6 shows the comparison of the estimated processing time and the measured processing time. Equation (3) is used to estimate the FE-GA core’s processing time. The accuracy of the estimation is more than 92%. The accuracy is calculated using Eq. (5). We think this accuracy level is
Table 4  FE-GA implementation of tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of contexts</th>
<th>PEs per a context</th>
<th>CRAMs per a context</th>
<th>Parallelism</th>
<th>Data amount (loop iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4224</td>
</tr>
<tr>
<td>Task 2</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>4224</td>
</tr>
<tr>
<td>Task 3</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>6144</td>
</tr>
<tr>
<td>Task 4</td>
<td>21</td>
<td>24</td>
<td>6</td>
<td>2</td>
<td>42048</td>
</tr>
<tr>
<td>Task 5</td>
<td>4</td>
<td>24</td>
<td>6</td>
<td>4</td>
<td>69632</td>
</tr>
<tr>
<td>Task 6</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>36864</td>
</tr>
</tbody>
</table>

Table 5  Processing time comparison of the task allocation.

<table>
<thead>
<tr>
<th>Task</th>
<th>Conventional method</th>
<th>Proposed method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocated core</td>
<td>Processing time (μs)</td>
</tr>
<tr>
<td>Data transfer</td>
<td>CPU —→ FE-GA</td>
<td>131.81</td>
</tr>
<tr>
<td>Task 1</td>
<td>FE-GA</td>
<td>2.19</td>
</tr>
<tr>
<td>Task 2</td>
<td>FE-GA</td>
<td>3.97</td>
</tr>
<tr>
<td>Task 3</td>
<td>FE-GA</td>
<td>6.16</td>
</tr>
<tr>
<td>Data transfer</td>
<td>FE-GA —→ CPU</td>
<td>831.35</td>
</tr>
<tr>
<td>Task 4</td>
<td>CPU</td>
<td>FE-GA</td>
</tr>
<tr>
<td>Task 5</td>
<td>CPU</td>
<td>650.32</td>
</tr>
<tr>
<td>Task 6</td>
<td>CPU</td>
<td>FE-GA</td>
</tr>
<tr>
<td>Data transfer</td>
<td>-</td>
<td>FE-GA —→ CPU</td>
</tr>
<tr>
<td>Total</td>
<td>1625.80</td>
<td>330.25</td>
</tr>
</tbody>
</table>

Table 6  Estimated vs. measured processing time in FE-GA.

<table>
<thead>
<tr>
<th>Task</th>
<th>Processing time (ms)</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Estimated</td>
<td>Measured</td>
</tr>
<tr>
<td>Task 1</td>
<td>2.04</td>
<td>2.19</td>
</tr>
<tr>
<td>Task 2</td>
<td>3.85</td>
<td>3.97</td>
</tr>
<tr>
<td>Task 3</td>
<td>5.67</td>
<td>6.16</td>
</tr>
<tr>
<td>Task 4</td>
<td>58.25</td>
<td>61.40</td>
</tr>
<tr>
<td>Task 5</td>
<td>74.88</td>
<td>79.97</td>
</tr>
<tr>
<td>Task 6</td>
<td>30.72</td>
<td>31.41</td>
</tr>
</tbody>
</table>

Table 7  Power and energy consumption.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Processing time (μs)</th>
<th>power (W)</th>
<th>Energy (W•s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-only</td>
<td>5554</td>
<td>1.21</td>
<td>$6.72 \times 10^{-3}$</td>
</tr>
<tr>
<td>CPU and accelerator (conventional method)</td>
<td>1626</td>
<td>1.30</td>
<td>$2.11 \times 10^{-3}$</td>
</tr>
<tr>
<td>CPU and accelerator (proposed method)</td>
<td>330</td>
<td>1.30</td>
<td>$0.43 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

Fig. 18  Processing speed comparison.

Table 8  Error in the task implementation on FE-GA.

<table>
<thead>
<tr>
<th>Task</th>
<th>Precision</th>
<th>Maximum error in FE-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3</td>
<td>17-bit</td>
<td>1</td>
</tr>
<tr>
<td>Task 4</td>
<td>17-bit</td>
<td>1.99</td>
</tr>
<tr>
<td>Task 5</td>
<td>16-bit</td>
<td>0.11 degrees</td>
</tr>
<tr>
<td>Task 6</td>
<td>20-bit</td>
<td>15</td>
</tr>
</tbody>
</table>

The processing time data in Tables 5, 6 and 7 are obtained by dividing the number of clock cycles by the frequency. The number of clock cycles are counted using a performance counter in the RP1 processor. To calculate power we measure the current from vdd to ground using a “current probe.” Then we multiply the current by the supply voltage to determine the power. The energy is calculated by using the power and processing time.

It is possible to use the proposed method in many CPU and accelerator cores. We can assign the operations in a loop to many tasks and run them in parallel in many accelerator cores. Moreover, if different tasks do not have data dependencies, we can run such tasks in parallel using many cores.

There is a trade-off between the performance and precision when using FE-GA. The CPU core in RP1 processor supports double-precision floating point (64-bit) calculations. The FE-GA supports only 16-bit fixed point calculations. We consider 2 kinds of errors in the Hog descriptor computation: the numerical errors and statistical errors. Numerical errors refer to the difference between the actual value and its rounded value. Table 8 shows the precision and the maximum numerical errors of tasks 3 to 6 in FP-GA.

processing time. In the proposed task allocation, the energy consumption reduced by 93% compared to CPU-only implementation. Compared to the conventional method, the energy consumption reduced by 79%.

The processing time data in Tables 5, 6 and 7 are obtained by dividing the number of clock cycles by the frequency. The number of clock cycles are counted using a performance counter in the RP1 processor. To calculate power we measure the current from vdd to ground using a “current probe.” Then we multiply the current by the supply voltage to determine the power. The energy is calculated by using the power and processing time.

It is possible to use the proposed method in many CPU and accelerator cores. We can assign the operations in a loop to many tasks and run them in parallel in many accelerator cores. Moreover, if different tasks do not have data dependencies, we can run such tasks in parallel using many cores.

There is a trade-off between the performance and precision when using FE-GA. The CPU core in RP1 processor supports double-precision floating point (64-bit) calculations. The FE-GA supports only 16-bit fixed point calculations. We consider 2 kinds of errors in the Hog descriptor computation: the numerical errors and statistical errors. Numerical errors refer to the difference between the actual value and its rounded value. Table 8 shows the precision and the maximum numerical errors of tasks 3 to 6 in FP-GA.

Table 7  Power and energy consumption.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Processing time (μs)</th>
<th>power (W)</th>
<th>Energy (W•s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-only</td>
<td>5554</td>
<td>1.21</td>
<td>$6.72 \times 10^{-3}$</td>
</tr>
<tr>
<td>CPU and accelerator (conventional method)</td>
<td>1626</td>
<td>1.30</td>
<td>$2.11 \times 10^{-3}$</td>
</tr>
<tr>
<td>CPU and accelerator (proposed method)</td>
<td>330</td>
<td>1.30</td>
<td>$0.43 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

It is possible to use the proposed method in many CPU and accelerator cores. We can assign the operations in a loop to many tasks and run them in parallel in many accelerator cores. Moreover, if different tasks do not have data dependencies, we can run such tasks in parallel using many cores.

There is a trade-off between the performance and precision when using FE-GA. The CPU core in RP1 processor supports double-precision floating point (64-bit) calculations. The FE-GA supports only 16-bit fixed point calculations. We consider 2 kinds of errors in the Hog descriptor computation: the numerical errors and statistical errors. Numerical errors refer to the difference between the actual value and its rounded value. Table 8 shows the precision and the maximum numerical errors of tasks 3 to 6 in FP-GA.

Table 8  Error in the task implementation on FE-GA.

<table>
<thead>
<tr>
<th>Task</th>
<th>Precision</th>
<th>Maximum error in FE-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3</td>
<td>17-bit</td>
<td>1</td>
</tr>
<tr>
<td>Task 4</td>
<td>17-bit</td>
<td>1.99</td>
</tr>
<tr>
<td>Task 5</td>
<td>16-bit</td>
<td>0.11 degrees</td>
</tr>
<tr>
<td>Task 6</td>
<td>20-bit</td>
<td>15</td>
</tr>
</tbody>
</table>
compared to their floating point implementation in CPU. In task 3, we perform 16-bit addition by neglecting the least-significant-bit. In task 4, we calculate the square-root of 17-bit data by neglecting the the least-significant-bit with a maximum error of 1.99. In task 5, we use 16-bit fixed-point calculation in FE-GA to calculate the boundaries of the bins. In this task, we get a maximum error of 0.11 degrees. In task 6, we perform 20-bit addition by neglecting the last 4 least-significant-bits. Therefore, the maximum error is 15. We can reduce these errors by assigning tasks to the CPU. However, that will increase both the data transfer time and the total processing time.

The numerical errors discussed above are maximum errors and have a low probability of occurrence in real world media processing. Therefore, we also calculated statistical error using many real-world images. The statistical error refers to the difference of the HOG descriptor vector component obtained by the CPU-only implementation and CPU + FE-GA implementation. According to the statistical results, vector components obtained by CPU + FEGA implementation have an error of 2.8% compared to those of CPU-only implementation. However, this error percentage can be different for the images with different contrasts and different light conditions. Therefore, the statistical errors depend on the input images.

The HOG descriptor is used for object detection that contains a machine learning process. Although the numerical and statistical errors have an effect on the object-detection-ratio, such effects can be reduced by using a good machine learning algorithm. Since machine learning is not in the scope of this paper it is not easy to discuss the performance decrease due to FE-GA implementation. However, since the statistical error is only 2.8%, we expect a high object-detection-ratio.

6. Conclusion

We proposed algorithm transformation to increase the freedom of the task allocation in the heterogeneous multicore processors. We show that the methods such as CORDIC algorithms, etc can be used successfully to change the accelerator-incompatible operations to accelerator-compatible operations. We have evaluated our method using HOG descriptor computation algorithm. The results shows that the proposed method reduces the total processing time and energy consumption by more than 79%.

In future works, we need a heuristic method for tasks allocation, since the exhaustive search increases the searching time exponentially with the number of tasks. Moreover, an automatic task partitioning method is also required for large size media processing applications.

References


Daisuke Okumura received the B.E. degree in Engineering from Sendai National College of Technology and M.S. degree in Information Sciences from Tohoku University, Japan, in 2008 and 2010 respectively. His research interests include Heterogeneous multi-core architectures and high-level design methodology for VLSIs.

Masanori Hariyama received the B.E. degree in electronic engineering, M.S. degree in Information Sciences, and Ph.D. in Information Sciences from Tohoku University, Sendai, Japan, in 1992, 1994, and 1997, respectively. He is currently an associate professor in Graduate School of Information Sciences, Tohoku University. His research interests include VLSI computing for real-world application such as robots, high-level design methodology for VLSIs and reconfigurable computing.

Michitaka Kameyama received the B.E., M.E. and D.E. degrees in Electronic Engineering from Tohoku University, Sendai, Japan, in 1973, 1975, and 1978, respectively. He is currently Dean and a Professor in the Graduate School of Information Sciences, Tohoku University. His general research interests are intelligent integrated systems for real-world applications and robotics, advanced VLSI architecture, and new-concept VLSI including multiple-valued VLSI computing. Dr. Kameyama received the Outstanding Paper Awards at the 1984, 1985, 1987 and 1989 IEEE International Symposiums on Multiple-Valued Logic, the Technically Excellent Award from the Society of Instrument and Control Engineers of Japan in 1986, the Outstanding Transactions Paper Award from the IEICE in 1989, the Technically Excellent Award from the Robotics Society of Japan in 1990, and the Special Award at the 9th LSI Design of the Year in 2002. Dr. Kameyama is an IEEE Fellow.