PERFORMANCE COMPARISON OF FPGA, GPU AND CPU IN IMAGE PROCESSING

Shuichi Asano, Tsutomu Maruyama and Yoshiki Yamaguchi

Systems and Information Engineering, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN
{asano,maruyama}@darwin.esys.tsukuba.ac.jp

ABSTRACT

Many applications in image processing have high inherent parallelism. FPGAs have shown very high performance in spite of their low operational frequency by fully extracting the parallelism. In recent micro processors, it also becomes possible to utilize the parallelism using multi-cores which support improved SIMD instructions, though programmers have to use them explicitly to achieve high performance. Recent GPUs support a large number of cores, and have a potential for high performance in many applications. However, the cores are grouped, and data transfer between the groups is very limited. Programming tools for FPGA, SIMD instructions on CPU and a large number of cores on GPU have been developed, but it is still difficult to achieve high performance on these platforms. In this paper, we compare the performance of FPGA, GPU and CPU using three applications in image processing; two-dimensional filters, stereo-vision and k-means clustering, and make it clear which platform is faster under which conditions.

1. INTRODUCTION

FPGAs have shown very high performance in many applications in image processing. However, recent CPU and GPU have also a potential for high performance for those problems. Recent CPU supports multi-cores, each supports improved SIMD instructions and executes up to 16 operations on 128b data in one clock cycle. Recent GPU supports a large number cores which run in parallel, and its peak performance outperforms CPU. In [9], we have compared the performance of CPU with quad-cores and FPGA using three applications in image processing; two-dimensional filters, stereo-vision [2][3][4], and k-means clustering [6][7][8]. In this paper, we compare the performance of GPU with FPGA and CPU using the same three problems.

The high performance of FPGA comes from its flexibility which makes it possible to realize the fully optimized circuit for each application, and a large number of on-chip memory banks which supports the high parallelism. Because of these features, FPGA can achieve extremely high performance in many applications in spite of its low operational frequency, though the designers need to tackle to minimize the number of operations and memory accesses in each unit to exploit more parallelism. The parallelism in SIMD instructions of CPU is limited, but the operational frequency of CPU is very high, and CPU is expected to show high performance in applications for which the cache memory works well. The size of the cache memory is large enough to store whole images in many image processing applications, and CPU can execute the same algorithms as FPGA in spite of the high memory bandwidth required by the algorithms. Therefore, the performance comparison of CPU and FPGA comes down to the trade-offs of the higher operational frequency and higher parallelism. The operational frequency of GPU is a bit slower than CPU (but faster than FPGA), but GPU supports a large number of cores (240 in our target GPU) which run in parallel, and its peak performance outperforms CPU. However, the cores are grouped, and the data transfer between groups is very slow. Furthermore, the size of the local memory provided for each group is very small. Because of these limitations, GPUs can not execute the same algorithms as FPGA in some application problems. In order to extract the high performance on GPU, we need to design the algorithm carefully so as to get around these limitations. Programming tools for GPU have been developed, but it is still difficult to achieve high performance on GPU as well as CPU and FPGA. We try to develop algorithms which give high performance on GPU for the three problems, and compare the performance with FPGA and CPU.

2. GRAPHICS PROCESSING UNIT

In this paper, we choose GPU (Graphics Processing Unit) from Nvidia for the evaluation, because the programming environment called CUDA[1] is supported for their products. Fig.1 show an overview of Nvidia GTX280, which is our target GPU. It consists of 10 thread processor clusters. A thread processor cluster has three streaming multiprocessors, eight texture filtering units, and one level-1 cache memory. Each streaming multiprocessor has one instruction unit, eight stream processors (SPs) and one local memory (16KB). Thus, GTX280 has 240 SPs in total. Eight SPs in a streaming multiprocessor are connected to one instruction unit. This means that the eight SPs execute the same instruction stream on different data (called *thread*). In

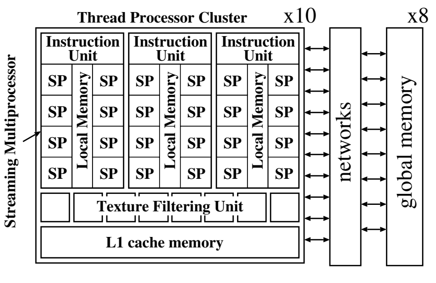

Fig. 1. An overview of Nvidia GTX280

order to extract the maximum performance of SPs by hiding memory access delay, we need to provide four threads for each SP, which are interleaved on the SP. Therefore, we have to provide 32 threads for each streaming multiprocessor, which amounts to 960 (=240 × 4) threads in total. Streaming multiprocessors are connected to large global memory through networks. The clock speed of processing units is 1296MHz, and 602MHz for other units.

Because of a larger number of cores, the peak performance of GPU outperforms CPU. However, the programming on GPU is even more tricky, because

1. access to the local memory from eight SPs in the same streaming multiprocessor is fast, but access to the global memory is despairingly slow (it takes several hundred clock cycles, though the throughput of the global memory is fast enough),
2. even for the local memory, eight SPs can not access arbitrary addresses in parallel (parallel access to only successive addresses (word boundary) is allowed),
3. the size of local memory is only 16KB (for 32 threads),
4. we have to provide a large number of threads, while minimizing sequential parts.

This programming is somewhat similar to the hardware design, because we need to make all stages of all pipelined circuits busy by extracting more parallelism, and we also need to minimize the number of memory accesses to off-chip memory banks in the hardware design.

3. TWO-DIMENSIONAL FILTERS

Let w be the radius of the filter, and $I(x,y)$ the value of a pixel at (x,y) in the image. Then, the computation for applying a non-separable filter to $I(x,y)$ becomes as follows.

$$S(x,y) = \sum_{dx=-w}^{w}\sum_{dy=-w}^{w} I(x{+}dx, y{+}dy)\cdot G(dx,dy)$$

where $G(dx,dy)$ is the coefficient on (dx,dy)

The computational complexity of this calculation is $O(w \times w)$. If $G(dx,dy)$ can be rewritten as $G_x(dx)\cdot G_y(dy)$ (this means that the coefficients in the filter can be decomposed along the x and y axes), the filter is called *separable*. The computational complexity of separable filters is $O(w)$, and the performance of CPU is faster than FPGA in practical range of w [9]. In this paper, we compare the performance of non-separable filters.

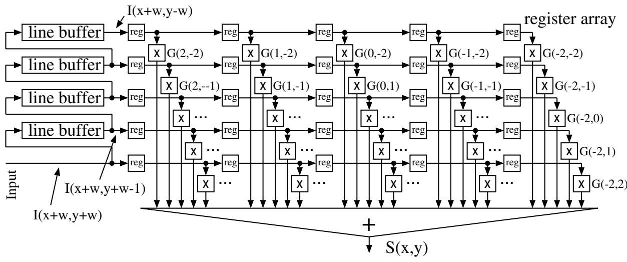

Fig. 2. Circuits for non-separable filters

Fig.2 shows a block diagram of a circuit for non separable filters when w=2, which is implemented on FPGA. In Fig.2, one pixel value $I(x{+}w, y{+}w)$ is given to the circuit every clock cycle, and sent to the register array. At the same time, data in the line buffers (the number of line buffers is $2w$) are read out in parallel (these data are $I(x{+}w, y{+}k)$ $k = -w{+}1, ..., w$), and given to the register array. The read-out data and $I(x{+}w, y{+}w)$ are held on the register array for $2w$+1 clock cycles and multiplied by $G(dx,dy)$. The products are summed up by an adder tree, and the result becomes $S(x,y)$. The read-out data and $I(x{+}w, y{+}w)$ are written back to the next line buffers for the calculation of $S(x, y{+}1)$. This circuit is fully pipelined, and can apply the filter to each pixel in one clock cycle (as its throughput). The number of operations is $(2w{+}1)\times(2w{+}1)$ for multiply operations, and $(2w{+}1)\times(2w{+}1){-}1$ for add operations. Using CPU with SIMD instructions, N pixels can be processed in parallel using N cores, and eight $I(x{+}dx, y{+}dy)$ can be multiplied by $G(dx,dy)$ in parallel in each core. In order to extract the maximum performance of GTX280, we need to provide at least 960 threads. In this problem, $S(x,y)$ can be computed independently for all pixels. Therefore, by computing 960 $S(x,y)$ in parallel (each of them is computed sequentially), we can fulfill all pipeline stages of GTX280. This implementation makes it easy to support arbitrary w using the same program. The number of operations in this computation is the same as FPGA and CPU.

4. STEREO VISION

In the stereo vision system, projections of the same location are searched in two images (I_r and I_l) take by two cameras, and the distance to the location is obtained from the disparity. In order to find the projections, a small window centered at a given pixel in I_r is compared with windows in I_l on the same line (epipolar restriction) in area-based matching algorithms. The sum of absolute difference (SAD) is widely used to compare the windows because of its simplicity. When SAD is used, the value of d in $[0, D{-}1]$ which minimizes the following equation is searched.

$$SAD_{xy}(x,y,d){=}\sum_{dx=-w}^{w}\sum_{dy=-w}^{w}|I_r(x{+}dx,y{+}dy){-}I_l(x{+}d{+}dx,y{+}dy)|$$

In this equation, $(2w{+}1)\times(2w{+}1)$ is the size of the window centered at a pixel $I_r(x,y)$, d is the disparity, and its range D decides how many windows in I_l (their centers are

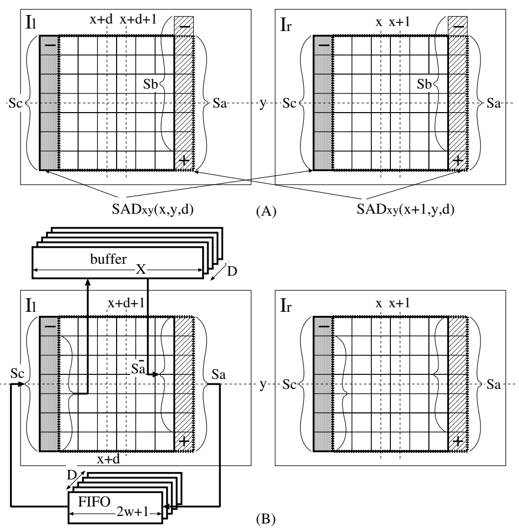
Fig. 3. A computation method of the stereo vision

$I_l(x{+}d, y))$ are compared with the window. The computational complexity of a naive implementation is $O(X{\times}Y{\times}w{\times}w{\times}D)$ when the size of the image is $X \times Y$.

Fig.3(A) shows how to compute $SAD_{xy}(x, y, d)$ in our approach[2]. $SAD_{xy}(x, y, d)$ can be rewritten as follows.

$$SAD_{xy}(x, y, d){=}\sum_{dx=-w}^{w} SAD_y(x{+}dx, y, d)$$

$$\text{where } SAD_y(x, y, d){=}\sum_{dy=-w}^{w} |I_r(x, y{+}dy){-}I_l(x{+}d, y{+}dy)|$$

Suppose that $SAD_{xy}(x, y, d)$ has been calculated, and we are now going to calculate $SAD_{xy}(x{+}1, y, d)$. In this case, $\sum_{dx=-w+1}^{w} SAD_y(x{+}dx, y, d)$ (*SAD* of white pixels in Fig.3) can be used for both $SAD_{xy}(x, y, d)$ and $SAD_{xy}(x{+}1, y, d)$. Therefore, $SAD_{xy}(x{+}1, y, d)$ can be calculated as $SAD_{xy}(x, y, d){+}SAD_y(x{+}1{+}w, y, d)$ (Sa in Fig.3) $-SAD_y(x{-}w, y, d)$ (Sc in Fig.3). Here, when $SAD_{xy}(x{+}1, y{-}1, d)$ was calculated in the upper line, $SAD_y(x{+}1{+}w, y{-}1, d)$ (Sb in Fig.3) was already calculated. $SAD_y(x{+}1{+}w, y, d)$ can be calculated by subtracting the absolute difference of the hatched pixel with '-' from $SAD_y(x{+}1{+}w,\ y{-}1, d)$, and adding that of the hatched pixel with '+' to the result. $SAD_y(x{-}w, y, d)$ was already calculated when $SAD_{xy}(x{-}2w, y, d)$ was calculated, and we can reuse it. Fig.3(B) shows how to manage those values. For each d in $[0, D{-}1]$, $SAD_y(x{+}1{+}w, y, d)$ is calculated by adding the absolute difference of the hatched pixel with '+' to buffer$[d][x{+}1{+}w]$ (S_a^- in Fig.3(B)). It is stored in FIFO and reused afterward. The absolute difference of the gray pixel with '-' is subtracted from the output of the FIFO ($SAD_y(x{-}w, y, d)$), and stored in buffer$[d][x{-}w]$. With this computation method, we can reduce the computational complexity to $O(X{\times}(Y{+}w){\times}D)$, though we need D buffers whose sizes are X and D FIFOs whose sizes are $2w{+}1$. The total number of operations per pixel is $2{\times}D$ for absolute differences, $4{\times}D$ for add/subtract operations, and $D{-}1$ comparisons for finding min of $SAD(x, y, d)$.

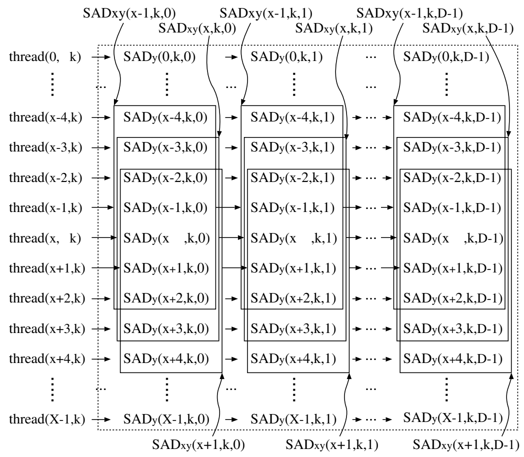
Fig. 4. A computation method of the stereo vision on GPU

There are two approaches for parallel processing of this method. First, D *SADs* ($SAD(x, y, d)$ $(d{=}0, D{-}1)$) can be calculated in parallel for each pixel. Second, N pixels can be processed in parallel, by dividing the image into N sub-images along y (or x) axis, though the total number of operations increases to $O(X{\times}(Y/N{+}w){\times}D{\times}N))$. With FPGA, we can compute D *SADs* in parallel, and find d which minimizes them on the fully pipelined unit. Therefore, its throughput becomes one pixel per clock cycle. This performance can be improved further by dividing an image into sub-images, and processing the sub-images in parallel. With CPU, we can also execute this computation method in parallel by processing each sub-image on each core. Up to 8 *SADs* can be computed in parallel using SIMD instructions in each core. GPU, however, can not execute this method, because D buffers and FIFOs required in this method can not be located in the local memory of GPU. GPU has large global memory, but the access delay to the memory is large, and can not be used for the buffers and FIFOs.

Fig.4 shows the basic computation method for GPU (a modification of a stereo vision algorithm implemented in FPGA in [5]) when $w{=}3$. In this method, the processing of each pixel becomes one thread. In Fig.4, X threads for X pixels on $y{=}k$ are shown. The X threads run synchronously in parallel in the same streaming multiprocessor sharing arrays located in the local memory. The procedure executed by thread(x, k, d) is as follows.

1. repeat the following step 2 - step 5 for $d{=}0, ..., D{-}1$.
2. calculate $SAD_y(x, k, d)$ by adding the absolute differences of $2w{+}1$ pixels sequentially.
3. store it in an array *sad_y*$[x]$.
4. calculate $SAD_{xy}(x, k, d)$ by adding *sad_y*$[x{+}dx]$ $(-w \le dx \le w)$ sequentially.
5. compare it with *sad_xy_min*$[x]$ (an array which stores the minimum SAD_{xy}), and update *sad_xy_min*$[x]$ and *d_min*$[x]$ (an array which stores d for the minimum) if it is smaller than *sad_xy_min*$[x]$.
6. output *d_min*$[x]$ as the disparity at (x, k).

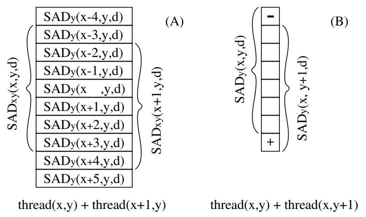

Fig. 5. Optimization methods of the stereo vision on GPU

In this method, one set of *sad_y*[], *sad_xy_min*[] and *d_min*[] is required. The computational complexity of this method is $O(X \times Y \times w \times D)$.

Fig.5 shows a technique to reduce the number of operations. First, in step 4, $2w+1$ SAD_y in *sad_y*[] are added sequentially. By merging two threads, thread(x, y) and $(x+1, y)$ as shown in Fig.5(A) ($w=3$), $SAD_{yx}(x+1, y, d)$ can be obtained by subtracting $SAD_y(x-3, y, d)$ from SAD_{xy} (x, y, d), and adding $SAD_y(x+4, y, d)$ to the result (not by adding $2w+1$ SAD_y). By merging more threads, we can reduce the computation time more. However, we need at least 32 threads for one streaming multiprocessor to fulfill the pipeline stages. Therefore, we can merge up to $X/32$ threads. Second, in step 2, the absolute differences of $2w+1$ pixels are added sequentially. By merging two threads, thread(x, y) and $(x, y+1)$ as shown in Fig.5(B), $SAD_y(x, y+1, d)$ can be obtained by subtracting the absolute difference of the pixel with '-' from $SAD_y(x, y, d)$ and adding that of the pixel with '+' to the result. The size of each array used in this method is X. When $512<X\leq 1024$, two sets of those arrays can be implemented in one local memory and two threads can be merged. However, in our experiments, the computation method which uses half of the local memory to cache pixel data shows better performance, and no local memory is left for merging the two threads. Then, the average number of operations per pixel is $(2w+1)\times D$ for absolute differences, $(2w+(2w+2\times(X/32-1))/(X/32))\times D$ for add/subtract operations, and $D-1$ for comparison to find the minimum, which is 3.5X for absolute differences, 2.05X for add/subtract operations and 1X for comparisons of the computation method used for FPGA when $w=3$ and $X=640$.

5. K-MEANS CLUSTERING

Given a set S of D-dimensional points, and an integer K, the goal of the k-means clustering is to partition the points into K subsets (C_i $(i = 1, K)$) so that the following error function is minimized.

$$E = \sum_{i=1}^{K} \sum_{x \in C_i} (x - center_i)^2$$

where $center_i = \sum_{x \in C_i} x \ / \ |C_i|$ (the mean of points in C_i). Fig.6 shows one iteration of the simple k-means clustering algorithm. First, squared distances to K centers are calculated for each point in the dataset and the minimum of them is chosen (the point belongs to the cluster which gives the minimum distance). Then, new centers are calculated from the points which belongs to each cluster. These operations are repeated until no improvement of E is obtained.

```
XOneIteration (Point Set $S$, Centers $Z$) {
  $E \leftarrow 0$;
  for each ($x \in S$) {
    $z \leftarrow$ the closest point in $Z$ to $x$;
    $z.weightCentroid \leftarrow z.weightCentroid+x$;
    $z.count \leftarrow z.count+1$;
    $E \leftarrow E+(x-z)^2$;
  }
  for each ($z \in Z$)
    $z \leftarrow z.weightCentroid/z.count$;
  return $E$;
}
```

Fig. 6. One iteration in the simple k-means algorithm

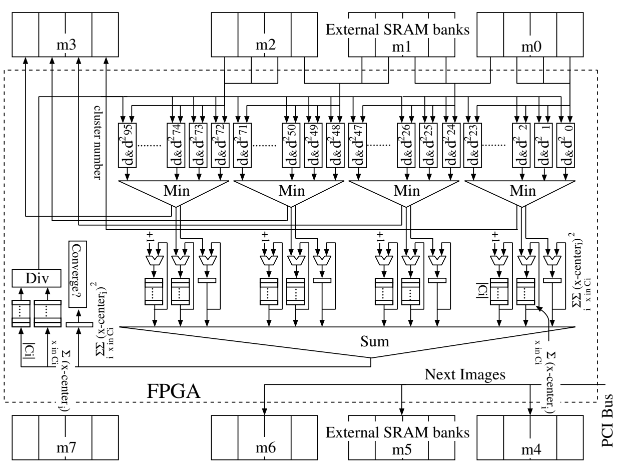

Fig. 7. A circuit for the simple k-means clustering algorithm

By applying the k-means clustering algorithm to color images, we can reduce the number of colors in the images to K while maintaining the quality of the images. Fig.7 shows a block diagram of a circuit for the simple k-means clustering algorithm for 24-bit full color RGB images. In Fig.7, pixels in one image are stored in three memory banks ($m0$, $m1$ and $m2$), and four pixels in the three memory banks are read out at the same time (data width of four pixels is 24b$\times$ 4, and the data width of three memory banks is 32b$\times$3) every clock cycle. The four pixels are processed in parallel on the fully pipelined circuit, and the results (cluster numbers for the four pixels) are stored in $m3$. After processing all pixels, four partial sums stored in on-chip memory banks are summed up to calculate new centers. While processing one image using four memory banks, next image can be downloaded to other memory banks. In order to find the closest center, squared Euclidean distances to K centers have to be calculated for each pixel. Let (x_R, x_G, x_B) be the value of a pixel. Then, its squared Euclidean distance to $center_i$ is

$$d^2 = (x_R - center_{i_R})^2 + (x_G - center_{i_G})^2 + (x_B - center_{i_B})^2$$

and, we need three multipliers to calculate one distance. In Fig.7, 96×3 units to calculate squared distance are used, which means that 96 squared Euclidean distances can be calculated in parallel (24 d^2 for each pixel because four pixels are processed in parallel). With CPU, we can process N pixels in parallel using N cores, and the distances to eight centers can be calculated in parallel in each core (the distance to each center is calculated sequentially).

As for GPU, we choose to process as many pixels as possible in parallel, but calculate the distances to K centers sequentially for each pixel, because this implementation makes it easier to write a program which can process arbitrary K. We need 960 threads for GTX280. In our current implementation, $Y{\times}m$ or $X{\times}n$ which is closer to $960{\times}l$ (m,n and l are integers) is chosen, and the image is divided into m sub-images ($(X/m){\times}Y$), or n sub-images ($X{\times}(Y/n)$). Then, the lines or columns in the sub-images are scanned in parallel. The threads run synchronously, and each thread executes the following procedure.

1. for each pixel p on the line or column, repeat the following steps.
2. calculate K distances sequentially for p, and find k_{min} which gives the minimum distance.
3. update four arrays (these arrays are located in each local memory, and used to calculate new centers for the next repetition).
   $R_{sum}[k_{min}]$ += x_R, $G_{sum}[k_{min}]$ += x_G, $B_{sum}[k_{min}]$ += x_B
   $count[k_{min}]$++
4. add the distance from p to $center_{k_{min}}$ to E.

After processing all pixels, $R_{sum}[]$, $G_{sum}[]$, $B_{sum}[]$ and $count[]$ in all streaming multiprocessors are summed up, and the new centers are calculated. E in all multiprocessors are also summed up, and the loop termination condition is checked. These summing up operations are executed using the global memory, which requires very long delay. In each streaming multiprocessor, 32 threads execute the procedure described above in parallel. This means that the four arrays and E are updated in parallel by those threads. However, this parallel access to shared arrays causes data access conflict on the local memory (called bank conflict), and are serialized, which probably decreases the performance.

6. EXPERIMENTAL RESULTS

For the evaluation, we use the following platforms.

1. Xilinx XC4VLX160
2. XFX GeForce 280 GTX 1024MB DDR3 standard and CUDA version 2.1
3. Intel Core 2 Extreme QX6850 (3GHz, quad-cores, 8MB L2 cache) and Intel C++ Compiler 10.0

The host computers have 4GB main memory. In the following comparison, the time to download images from main memory is not included, and the performance is the average of 1000 runs. In CPU, four threads runs in parallel on the quad-cores. The operational frequency of FPGA is fixed to 100MHz (though it is faster than that) to simplify the comparison.

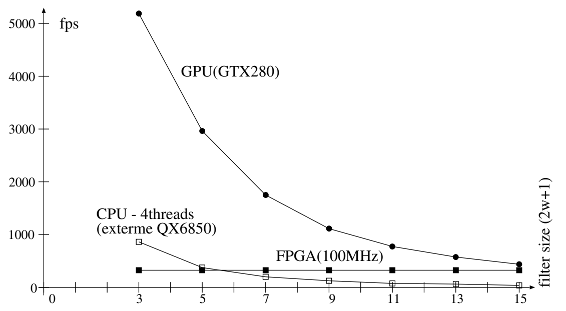

Fig. 8. Performance of two-dimensional filters

Fig.8 compares the performance of the platforms when a $(2w{+}1){\times}(2w{+}1)$ filter is applied to $640{\times}480$ pixel grayscale image. GPU is the fastest for all tested filter size. In this problem, filters can be applied to each pixel in the image independently without using shared variables. So, GPU can show its best performance. When, the filter size is $15{\times}15$, 225 pixels are processed in parallel in FPGA. The parallelism in GPU is 240. Considering the operational frequency of GPU, and the fine-grained parallelism in FPGA, the close performance seems to be reasonable. CPU is faster than FPGA when the filter is smaller than $5{\times}5$. When the filter size is 15×15, its performance (43 fps) is 1/8 of FPGA.

Fig.9 compares the performance for the stereo vision of $640{\times}480$ pixel grayscale images. In Fig.9, the size of window is 7×7. In this problem, FPGA is the fastest, and GPU is the slowest. As for GPU, two lines are shown, one is for the computation method proposed in Section 4, and another for the non-optimized method, in which $(2w{+}1){\times}(2w{+}1)$ absolute differences are calculated and then summed up for computing each $SAD_{xy}(x, y)$. In this non-optimized method, each pixel can be processed independently without using shared variables, and all processing units in GPU work most efficiently. However, the computational complexity of this method is high, and the performance is the worst. When $D{=}240$, its performance is 1/30 of FPGA, but the number of absolute difference operations is 25X. Therefore, it can be considered that GPU can execute almost the same number of operations in a unit time as FPGA, when all units work busily. Using the proposed method, we can reduce the computational complexity, but the number of operations is still more than twice as described in Section 4, and even worse, some parts in this method can not be parallelized because of the access conflict on the local memory (access delay to the global memory is avoided by caching pixel data in the local memory). The performance of CPU by 4 threads is about 3.6 times of 1 thread, and faster than 30 fps when $D \leq 224$. When $D{=}240$, its performance is 2X of GPU, but 1/12 of FPGA. The performance of FPGA becomes stepwise. In FPGA, a window in I_l can be compared with 242 windows in I_r in parallel (this is limited by the number of block RAMs). Therefore, when D is less than 121, two windows in I_l can be compared with 121 windows in I_r in parallel respectively.

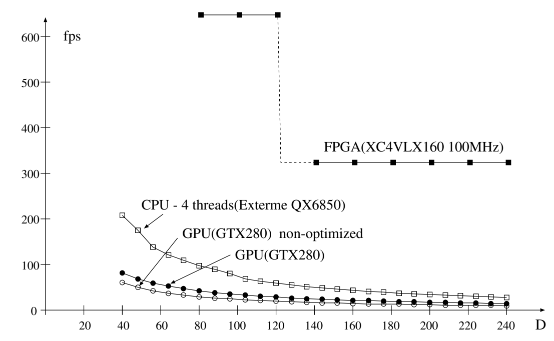

Fig. 9. Performance of the stereo-vision

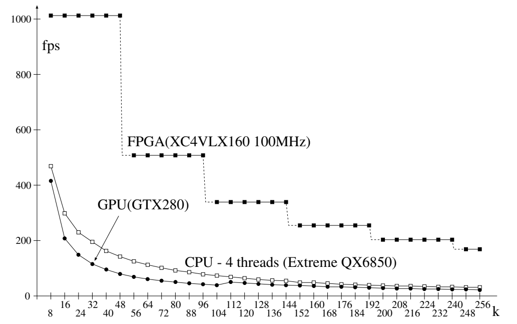

Fig. 10. Performance of the k-means clustering algorithm

Fig.10 compares the performance for the k-means clustering of 768×512 pixel color image. Fig.10 shows the performance of one iteration, because the performance of the k-means clustering depends on the number of iterations which varies from image to image, but the computation time of one iteration is decided just by K and the image size. In order to simplify data assignment to threads in GPU, only 768 threads are provided in the current implementation and, in each thread, 512 pixels are processed sequentially. Therefore, the performance can be improved by 25% by equally assigning pixels to 960 threads. However, the performance of GPU is still worse than CPU. The programs executed on GPU and CPU are almost the same, but in GPU, in order to hide the access delay to pixel data in the global memory, the loop in each thread is unrolled by hand, and next 32 pixels are loaded while processing current 32 pixels. We are now trying to make the reason of the low performance clear, but it probably comes from the access conflict for updating the four arrays in each streaming processor, and large latency to the global memory when summing up all the arrays in all streaming multiprocessors using the global memory. The performance of CPU by 4 threads is about 3.7 times of 1 thread. When D=48, the performance is about 1.8X of GPU and 1/7 of FPGA. The performance of FPGA becomes stepwise again, because the distances up to 48 clusters can be calculated in parallel, and for more centers, the same operations are repeated.

7. DISCUSSION AND FUTURE WORKS

We have compared the performance of GPU with FPGA and CPU (quad-cores) using three simple problems in image processing. GPU has a potential for achieving almost the same performance with FPGA. The number of cores in GTX280 is 240. With one XC4VLX160, 15×15 multiply and add operations (filter), 241 $SAD_{xy}(x, y)$ (stereo vision) and 48×4 distances (k-means clustering) can be computed in parallel. Considering the trade-offs between the operational frequency of GPU (more than 10 times faster), and the fine-grained parallelism in FPGA, this seems to be a natural consequence. However, GPU can show its potential only for naive computation methods, in which all pixels can be processed independently. For more sophisticated algorithms which use shared arrays, GPU can not execute those algorithms because of its very small local memory, or can not show good performance because of the memory access limitation caused by its memory architecture. GPU is slower than CPU in those algorithms (it may be possible to realize much better performance if we can find algorithms which can get around the limitations, but we could not find them). The performance of CPU is 1/12 - 1/7 of FPGA, which means that CPU with quad-cores can executes about 1/10 operations of FPGA in a unit time (the same algorithms are executed on CPU and FPGA). The performance of FPGA is limited by the size of FPGA and the memory bandwidth. With a latest FPGA board with DDR-II DRAM and a larger FPGA, it possible to double the performance by processing twice the number of pixels in parallel.

We have the following issues which have to be considered. We have compared the performance using only three problems. The performances of the programs on GPU and CPU are not fully tuned up. In the comparison, power consumption and costs are not considered.

8. REFERENCES

[1] http://www.nvidia.com/object/cuda_home.html

[2] H. Niitsuma, T. Maruyama, “Real-time Generation of Three-Dimensional Motion Fields”, FPL 2005, pp. 179–184

[3] A.Darabiha, et.al., “Video-rate stereo depth measurement on programmable hardware”, Computer Vision and Pattern Recognition, 2003, pp.203-210.

[4] J.Diaz, et.al, “FPGA-based real-time optical-flow system”, IEEE Transactions on Circuits and Systems for Video Technology (TCSVT), 2006, Vol.16, Issue.2, pp.274-279

[5] Y. Miyajima T.Maruyama, “A Real-Time Stereo Vision System with FPGA” FPL 2003, pp.448-457

[6] T. Saegusa, T. Maruyama, “An FPGA implementation of real-time K-means clustering for color images”, Journal of Real-Time Image Processing, Vol.2, No.4, 2007, pp.309-318

[7] M. Estlick, M. Leeser, J. Theiler and J. J. Szymanski “Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware”, FPGA 2001, pp 103 –110.

[8] B.Maliatski, O.Yadid-Pecht, “Hardware-driven adaptive k-means clustering for real-time video imaging”, IEEE TCSVT, Vol.15, Issue.1, 2005, pp. 164–166

[9] T.Saegusa, T.Maruyama, Y.Yamaguchi, “How fast is an FPGA in image processing?”, FPL 2008, pp.77-82.