Coarse Grain Mapping Method for Image Processing on Fine Grain Cellular Processor Arrays

Bin Wang and Piotr Dudek
School of Electrical and Electronic Engineering
University of Manchester
Manchester, United Kingdom
bin.wang-2@postgrad. manchester.ac.uk
p.dudek@manchester.ac.uk

***Abstract*—This paper introduces a mapping method for adding a coarse grain (multiple pixels per processor) processing mode to massively parallel cellular processor arrays. The main motivation is to provide the fine grain pixel-parallel processor array with the ability of processing images with higher resolution than the array itself, in a way that is transparent to the programmer. The proposed method accomplishes the mapping work entirely during the code compilation process, which has four main advantages. Firstly, there is no extra overhead during processing. Secondly, the source code for fine grain mode can be used in coarse grain mode without modification. Thirdly, the proposed method does not introduce any restrictions of the number of pixels stored in a processing element. Finally, the proposed method is easy to implement, as it does not require any modifications to the hardware design of the pixel-parallel processor array or its controller, but only to the software compiler. The mapping method and its software implementation are presented in this paper.**

I. INTRODUCTION

The current research into massively parallel cellular processor arrays (CPAs), consisting of a great number of Processing Elements (PEs), indicates that they can provide significant advantages, in terms of both performance and power consumption, in highly computationally demanding image processing applications [1-6]. For a typical fine grain cellular processor array considered here, the PEs are arranged as a 2D mesh, as shown in Fig.1(a). Each PE contains several *registers* as its local memory, ALU and other processing and control circuits. Due to the constraints of the silicon area and the fact that a large number of PEs is required, the number of registers in each PE is limited to a small number, e.g. 64 bit memory in [5], 34 bit memory in [3], 4 bit memory in [6], 5 analog and 4 binary registers in [2] and 9 analog registers in [4]. Although pixel parallel processor arrays provide significant performance advantages, the array sizes typically implemented on a single chip are relatively small, e.g. 128x128 [4], 176x144 [2], restricting the size of the image that can be processed. Therefore, some processor arrays are designed to process images that have higher resolution than the processor array itself, by storing and processing multiple pixels (e.g. 8x8 [7], 4x1 [9]), in a single PE, as shown in Fig.1(b). The emergence of 3D integration technology which

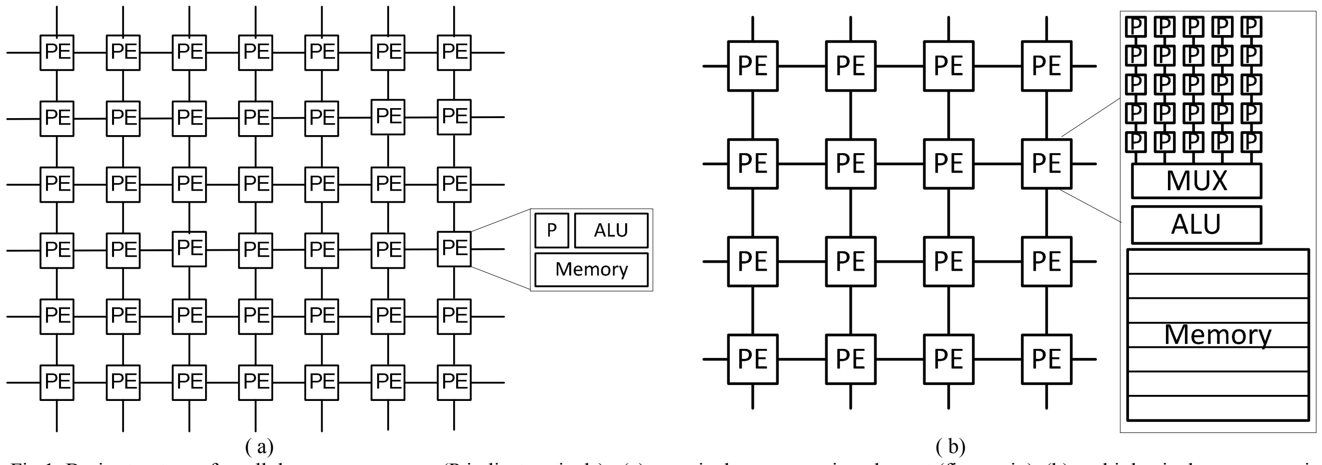


Fig.1. Basic structure of a cellular processor array (P indicates pixels): (a) one pixel per processing element (fine grain), (b) multiple pixels per processing element (coarse grain)

involves stacking silicon wafers and vertical interconnections using through-silicon-vias, potentially enhances the memory capacity of a CPA, providing the capability of storing a block of pixels in a PE, while maintaining the relatively high resolution of the processor array [8].

One method for a pixel-parallel cellular processor array processing an image with higher resolution relies on splitting the image into small windows to fit the size of the processor array. The image can then be processed window by window. However, as a concomitant, the boundary effects have to be carefully considered. Alternatively, solutions that allow processing of multiple pixels per PE could be employed. For example, a method mentioned in [9] stores four pixels in a PE, which enables a 32x128 processor array to process 128x128 images. However, such schemes usually involve specially designed hardware and elaborate application source code (as compared with one based on simple pixel-parallel array operations). Also, the hardware mapping method may provide restrictions on the number of pixels processed in each PE.

In this paper we will refer to a *fine grain* processing mode when a cellular processor array is allocated one pixel per PE, and a *coarse grain* processing mode when a processor array is used to process multiple pixels per PE.

This paper introduces a Coarse Grain Mapping Method (CGMM) that provides fine grain cellular processor arrays with a coarse grain processing mode, introducing a systematic software approach to processing large images on a small CPA. The proposed method has a number of advantages. Firstly, the mapping procedure itself does not produce any extra computational overhead for the processor array. Secondly, the source code written for the cellular processor array working in the fine grain mode can be used in coarse grain mode to process larger images. Thirdly, the proposed method is able to achieve much more flexibility, processing any form of pixel distributions, e.g. 2x2, 4x4, 2x8 pixels per PE, which is only limited by the number of registers in each PE. Finally, this function is accomplished purely by the compiler at compile time. It does not require any modification to the hardware, but only software compilers.

II. Coarse Grain Mapping Method

To process an image with a higher resolution than the processor array itself, a method to store the image in the cellular processor array needs to be established. To simplify the explanation, four assumptions are made: (1) The size of the PE array is NxM; (2) each PE has several registers $\mathrm{Ri}, \mathrm{i} \in \{1, 2, 3, 4 \ldots \mathrm{N_R}\}$; (3) Each PE can communicate with its neighbours; (4) Each PE has the ability to implement basic operations such as add, subtract, load, etc.

There are two ways of storing a bigger image (e.g. a 256x256 image) in a smaller processor array (e.g. 128x128) while maintaining the spatial distribution of pixels amongst processors. The first one is that each register stores a sub window of the image, e.g., register R1 stores the upper left part of the original image, as shown in Fig.2(b). The second one is that each PE stores a sub window of the image, e.g. PE(0,0) of processor array stores Pixel(0,0) of original image

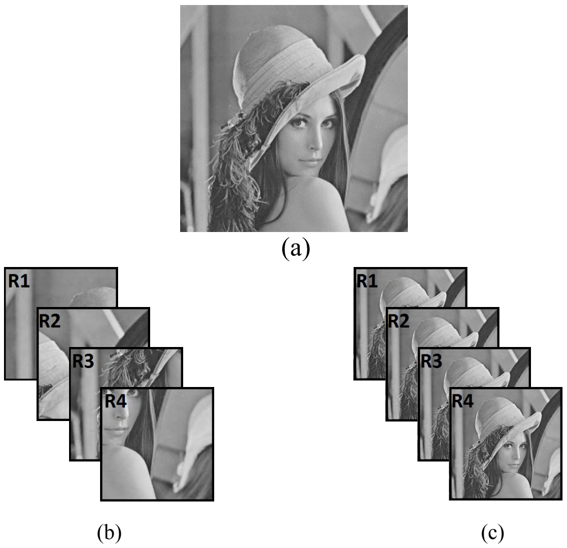


Fig.2. Two different pixel distribution methods: (a) is the original 256x256 image that needs to be distributed into 4 128x128 registers, e.g. R1, R2, R3 and R4; (b) shows the method that each register stores a sub window of original image; (c) shows the method that each PE stores a sub window of original image.

in register R1, Pixel(0,1) in R2, Pixel(1,0) in R3 and Pixel(1,1) in R4, as shown in Fig.2(c).

The basic requirement for a functional storing method is that each pixel can access its neighbours conveniently (without any long distance communication). If each register stores a sub window of the image, the original pixel connections at the sub window boundaries are lost. In other words, the boundary pixels cannot find their neighbours in their own PEs or the PEs around, e.g., the Pixel(127, 127) is stored in register R1 of PE(127, 127), while its neighbour Pixel(127,128) is stored in R2 of PE(127, 0). These boundary issues, appearing at the connections of the four sub-windows, require long distance communication to solve. It is cumbersome at best and carries a large overhead on a typical cellular processor array, which may lack the necessary long distance communication mechanisms. Consequently, the method whereby each PE is storing a small window of original image (Fig.2c) is used in the CGMM, because it can ensure that the neighbours of a pixel are always stored in a neighbouring PE. It should be noted here, that the size of the images that can be stored in the processor array is only limited by the number of PE registers. Also, the distribution method is completely flexible. For example, the group of pixels stored in a PE could be distributed as 1x3, 2x1 or 4x4 arrays to fit the size of input images of 128x384, 256x128 or 512x512 pixels respectively. The CGMM in general does not put any restriction on the size of the processor array, the number of registers, and the pixel distribution method. However, the hardware constraints will limit the available pixel distributions in practical cases.

After the processed image is distributed into registers, pixel communication with the neighbours becomes a critical issue. For example, assume four pixels stored in one PE are distributed as a 2x2 array into registers R1, R2, R3 and R4, as

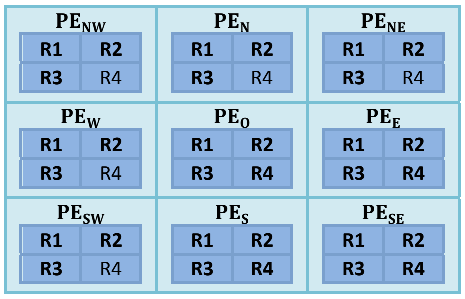
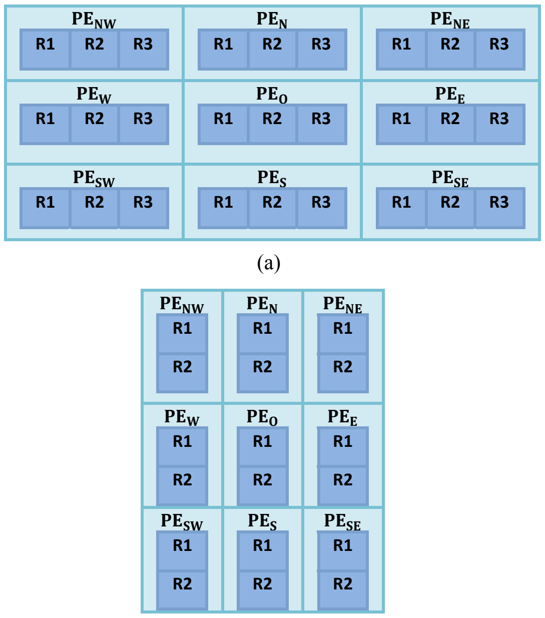


Fig.3. 2x2 pixel distribution map. The pixels are stored in registers R1, R2, R3 and R4; PE_E, PE_W, PE_N, PE_S indicate the four direct neighbours of PE_O; $PE_{NE}, PE_{NW}, PE_{SW}, PE_{SE}$ indicate the four indirect neighbours of PE_O.

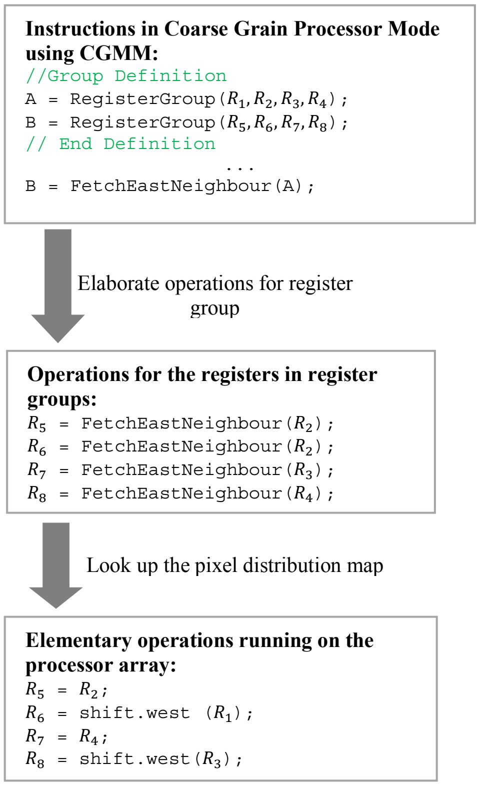


(b)

Fig.4. Example pixel distribution maps: (a) 1x3, (b) 2x1.

shown in Fig.3. When communicating with the neighbours, the western neighbour of pixel R2 in $\mathrm{PE_O}$ is pixel R1 in $\mathrm{PE_O}$, while the western neighbour of pixel R1 in $\mathrm{PE_O}$ is pixel R2 in $\mathrm{PE_W}$. For different pixel distribution methods, the rules of fetching pixels' neighbours are completely different. In order to maintain the flexibility of CGMM, a universal neighbour fetching method is required.

Once the pixel distribution method is confirmed before compilation, a Pixel Distribution Map (PDM) is created automatically according to the pixel distribution method (such as the one shown in Fig.3 for 2x2 sub-window distribution). A PDM, containing all the pixels in the communication range of $\mathrm{PE_O}$, provides a neighbour fetching index. When there is a pixel communication, the compiler always checks in the PDM to locate its neighbours first, and then uses the returned parameters to fetch these neighbours. For example, assume an image A is stored in registers R1, R2, R3 and R4 as shown in Fig.3, and the task is to fetch each pixel's eastern neighbour and then store the new image B in registers R5, R6, R7 and R8



Instructions in Coarse Grain Processor Mode using CGMM:

```
//Group Definition
A = RegisterGroup(R1,R2,R3,R4);
B = RegisterGroup(R5,R6,R7,R8);
// End Definition
                 ...
B = FetchEastNeighbour(A);
```

Elaborate operations for register group

Operations for the registers in register groups:

```
R5 = FetchEastNeighbour(R2);
R6 = FetchEastNeighbour(R2);
R7 = FetchEastNeighbour(R3);
R8 = FetchEastNeighbour(R4);
```

Look up the pixel distribution map

Elementary operations running on the processor array:

```
R5 = R2;
R6 = shift.west (R1);
R7 = R4;
R8 = shift.west(R3);
```

Fig.5. Instruction compilation process for fetching a register's eastern neighbour for the example distribution in Fig.3.

respectively. The whole process is illustrated as follows. For register R1, the compiler would first locate register R1 of $\mathrm{PE_O}$ in PDM. Then it will check the eastern neighbour of register R1 of $\mathrm{PE_O}$, record the register's name (e.g. register R2) and which PE it belongs to in the PDM (e.g. $\mathrm{PE_O}$). According to these two parameters, each register (R1, R2, R3 or R4) will find its eastern neighbour automatically, and its value will be loaded to their target registers respectively, e.g. R5 loads the value stored in R2 of $\mathrm{PE_O}$; R6 loads the value in R1 of the $\mathrm{PE_E}$; R7 loads the value stored in R4 of $\mathrm{PE_O}$; R8 loads the value stored in R3 of $\mathrm{PE_E}$. As a result, the image shifted by one pixel to the west is stored in registers R5, R6, R7 and R8.

When processing the images with different sizes (e.g. 128x384 or 256x128), new PDMs, e.g. as shown in Fig.4, are generated according to the image sizes while the neighbour fetching procedure remains the same. This feature makes the proposed mapping method more flexible than other mapping methods, such as the one used in [9] which restricts the pixels in each PE to a 4x1 array.

An instruction, such as fetching every pixel's eastern neighbour, is elaborated to several elementary operations (four in the previous example) when operating in coarse grain mode. The basic requirement is that this process should be abstracted away from programmers. A programmer could write code representing pixel-parallel (fine grain) array operations such as B = A (copy pixel array A to pixel array B) or B =FetchEastNeighbour(A) (copy array A to B while shifting it one pixel to the west), while in the coarse grain mode, this one instruction should be automatically translated

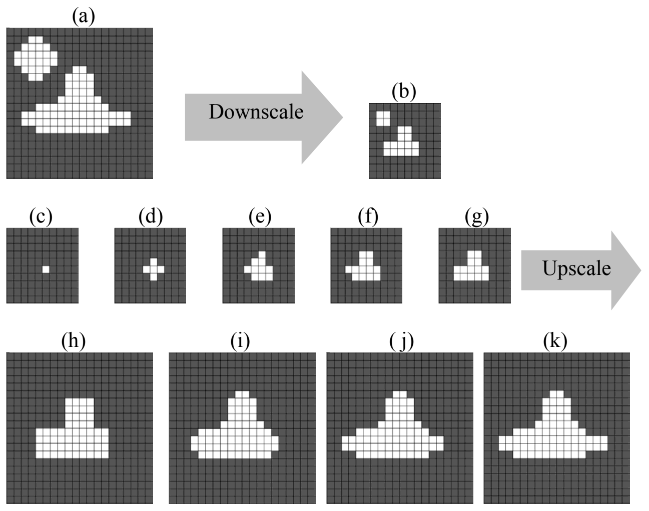


Fig.6. Multi-scale propagation process. (a) is the full resolution image; (b) is the coarse resolution image; (c), (d), (e), (f), and (g) indicate the coarse resolution propagation process, while (g) is the result of the coarse resolution propagation; (h), (i), (j) and (k) demonstrate the full resolution propagation process; (k) is the final result of the multi-scale propagation result in this example.

to *n* different elementary operations for *n* pixels stored in each PE, according to the pixel distribution method. To achieve this, a concept of a *register group* is introduced. A register group is defined as a group of registers that are always processed by the same instruction. Based on this concept, the four elementary operations used to fetch eastern neighbour of image A in the example above can be represented by one instruction.

A single instruction, e.g. B = FetchEastNeighbour(A), generates different ICWs for a particular CPA according to different processing modes. If the CPA is working in fine grain mode, this single instruction will translate into one elementary operation, which loads a register into another register, while shifting that register to west. If the CPA is working in coarse grain mode, this instruction will translate into several elementary operations, 4 in the previous example, using CGMM, which load several registers into other registers respectively, according to the given distribution method, as shown in Fig.5. The ICWs can then be generated for these elementary operations, according to the ICW elaboration rules for a particular CPA. Due to the fact that CGMM is a pre-processing mapping method implemented at the compiling time, it does not require any hardware modification, and can be used for any fine grain cellular processor array.

III. TRIGGER-WAVE PROPAGATION IN THE COARSE GRAIN MODE

A trigger-wave propagation [10], widely used in image processing algorithms, could be directly applied to coarse grain mode using CGMM if a synchronous, iterative propagation method is used. However, taking advantage of the fact that each PE can store more than one pixel in coarse grain mode, the processing speed of synchronous trigger-wave propagation can be boosted by a multi-scale propagation process. Instead of propagating binary waves in the full resolution image, propagation can start in the coarse resolution image to obtain an approximate result first. Furthermore, this also enables asynchronous propagation on the coarse scale, and then iterative synchronous full resolution propagation could be used to obtain an accurate boundary.

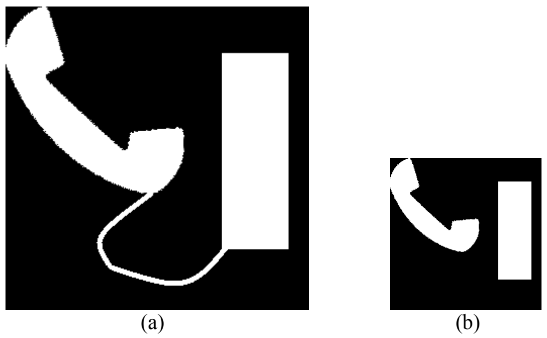


Fig.7. An example of images where the coarse propagation can only provide limited speedup. (a) is the original full resolution image, (b) is the coarse image. Because of downscaling, the line connecting two regions disappears.

Following is an example of this process (Fig.6). Assuming that a 20x20 image is processed on a 10x10 CPA using propagation, and the original image is stored in registers R1, R2, R3 and R4, the process is elaborated as follows. (1) A coarse image (stored in register R5) with the same resolution as the processor array is created (Fig.6(b)). The value of each pixel in this coarse image is the result of the AND operation of the 4 pixels of the original image stored in this PE. (2) This coarse image is used to carry out the propagation (synchronous or asynchronous) (Fig.6(c-g)). Due to the downscaling, this propagation will not provide an accurate result. (3) The coarse image is upscaled to the original resolution (Fig.6(h)). (3) Several single pixel propagation steps (operating on the original images using CGMM) are used to obtain the accurate result (Fig.6(i-k)). Step (3) will terminate when there is no pixel being propagated during a single step of synchronous propagation.

Although this multi-scale propagation process is faster compared with normal synchronous propagation, the increase can be small in some particular situations. For example, as can be seen from Fig.7 when an object consists of two large regions connected by a thin line and the propagation is triggered within one region, only a part of the object can be reconstructed through the coarse image propagation because after downscaling the thin line will disappear. A large number of synchronous propagation step needs to be used to propagate through the other region via the connecting line, which decreases the speed efficiency. A possible solution to this problem could be a speculative coarse propagation within disjoint regions, but this is a subject of further research beyond the scope of this paper.

```
    !include('apron.aps')
    !include('apron.io.aps')

(1) !include('apron.FineToMassive.aps')
(2) CGMM.createmap(1,2)
    A = CGMM.registergroup(R1, R2)
    B = CGMM.registergroup(R3, R4)
(3) C = CGMM.registergroup(R5, R6)
    D = CGMM.registergroup(R7, R8)
    E = CGMM.registergroup(R9, R10)

    #start
    A = io.image.open('lena.bmp')
    B = shift.east(A, 0)
    C = A + B
    B = shift.west(C, 0)
    E = B + C
    C = shift.north(E, 0)
    B = shift.south(E, 0)
    E = B - C
    B = abs(E)
    C = shift.north(A, 0)
    D = A + C
    C = shift.south(D, 0)
    E = C + D
    D = shift.east(E, 0)
    C = shift.west(E, 0)
    E = D - C
    C = abs(E)
    D = B + C
    jump(#start)
```

(a)

```
#start
//A = io.image.open('lena.bmp')
PIX = io.image.open('lena.bmp')
CMGG.distribute(PIX, R1, R2)
//B = shift.east(A, 0)
R3 = shift.east(R2, 0)
R4 = R1
//C = A + B
R5 = R1 + R3
R6 = R2 + R4
//B = shift.west(C, 0)
R3 = R6
R4 = shift.west(R5, 0)
//E = B + C
R9 = R3 + R5
R10 = R4 + R6
//C = shift.north(E, 0)
R5 = shift.north(R9, 0)
R6 = shift.north(R10, 0)
//B = shift.south(E, 0)
R3 = shift.south(R9, 0)
R4 = shift.south(R10, 0)
//E = B - C
R9 = R3 - R5
R10 = R4 - R6
//B = abs(E)
R3 = abs(R9)
R4 = abs(R10)
//C = shift.north(A, 0)
R5 = shift.north(R1, 0)
R6 = shift.north(R2, 0)
//D = A + C
R7 = R1 + R5
R8 = R2 + R6
//C = shift.south(D, 0)
R5 = shift.south(R7, 0)
R6 = shift.south(R8, 0)
//E = C + D
R9 = R5 + R7
R10 = R6 + R8
//D = shift.east(E, 0)
R7 = shift.east(R10, 0)
R8 = R9
//C = shift.west(E, 0)
R5 = R10
R6 = shift.west(R9, 0)
//E = D - C
R9 = R7 - R5
R10 = R8 - R6
//C = abs(E)
R5 = abs(R9)
R6 = abs(R10)
//D = B + C
R7 = R3 + R5
R8 = R4 + R6
jump(#start)
```

(b)

Fig.8. An example APRON source code: (a) Sobel edge detector; (b)its elementary operations generated using CGMM. The overhead part of the program is shown in the red frame in (a). In this example, each PE stores 1x2 pixels. When the pixel distribution method changes, only the lines in the frame need to be changed. When the program is running in coarse grain mode, after compilation, the source code between #start and #end will translate to the elementary operations shown in (b). The code after #start in (a) is the same as the code used in fine grain mode (which would use a straightforward mapping, e.g. A = R1, B = R2, C = R3, D =R4, E=R5).

IV. Software Implementation and Simulation Result

The CGMM was implemented using APRON software [11] which provides an integrated development environment for code development and cellular processor array emulation. Simulation results were obtained based on the APRON implementation of ASPA [5] processor model, with extended local memory. An example program of Sobel edge detector is shown in Fig.8(a) (detailed information about Sobel edge detector can be found in [12]). The source code for the coarse grain mode is only adding several lines (shown in the frame in Fig.8(a)), which define: (1) the processor array works in coarse grain mode; (2) custom definitions of the pixel distribution method for each PE; (3) five register groups. The program can be converted back to fine-grain mode by simply

Table I. Number of elementary operations that are generated for different algorithms running on a 128x128 fine grain processor array (ASPA model modified) in different processing modes for images with different sizes

	Sobel's Edge Detector	Game of Life	Skeletonization	Hole Filling
128x128 Processor Array in Fine Grain Mode	17	71	33	13
128x128 Processor Array in Coarse Grain Mode & 256x256 Images	68	230	117	43
128x128 Processor Array in Coarse Grain Mode & 512x256 Images	136	442	229	83
128x128 Processor Array in Coarse Grain Mode & 512x512 Images	272	866	453	163

deleting this additional code. When the source code is running in coarse grain mode using CGMM, the actual elementary operations are shown in Fig.8(b). Note that after compilation the increase of instruction length is completely caused by the increase of the resolution of processed images.

The elementary operation counts for several programs executing on a 128x128 processor array processing various sizes of images are shown in the Table I. As one can easily calculate from Table I, the number of elementary operations increases proportionally to the image size, i.e. the CGMM does not introduce any overhead for the hardware. In fact, some optimisations of the common sections of the code are possible, that further reduce the length of a program in the coarse grain mode (as can be seen in Game of Life, Skeletonization and the Hole Filling algorithms). It can also been seen from Fig.8(b), that not all the instructions are translating to two elementary operations.

V. Conclusion And Future Works

In this paper, a simple coarse grain mapping method is introduced. Utilizing the proposed method, a pixel-parallel cellular processor array can process bigger images without any change of the hardware, assuming the processor array has the adequate local memory resources. All the mapping work is automatically completed by the compiler. Programs written for fine grain mode can be easily used in the coarse grain mode without modifications. The results indicate that the proposed method does not introduce any extra overhead at the processing stage. The only requirement is a sufficient number of registers in each PE to handle the operations on multiple pixels per PE.

References

[1] A. Lopich, D. Barr, B. Wang and P. Dudek, "Real-time image processing on ASPA2 vision system", IEEE International Symposium on Circuits and System, ISCAS 2011, pp. 1989, 2011;

[2] A. Rodrigurez-Vazquez, E. Dominguez-Castro, F. Jimenez-Garrido, S. Morillas, J. Listan, L. Alba, C. Utrera, S. Espejo and R. Romay, "The Eye-RIS COMS vision system" in Analog Circuit Design, A. Roermund and M. Steyaert, Eds. Spinger, 2008, pp.15-32;

[3] J. Poikonen, M. Laiho and A. Paasio, "MIPA4K: a 64x64 cell mixed-mode image processor array", IEEE International Symposium on Circuits and System 2009, pp. 1931-1931, 2009;

[4] P.Dudek and S. Carey, "General-purpose 128x128 SIMD processor array with integrated image sensor", Electronics Letters, Vol. 42, Issue. 12, pp. 678-679, 2006;

[5] A. Lopich and P. Dudek, 'ASPA: focal plane digital processor array with asynchronous processing capabilities', ISCAS 2008, pp. 1592-1595, 2008;

[6] W. Miao, Q. Lin, W. Zhang and N. Wu, "A programmable SIMD vision chip for real-time vision applications", IEEE Journal of Solid-State Circuits, Volume 43, Issue 6, pp. 1470-1479, 2008;

[7] P. Foldesy, A. Zarandy, C. Rekeczky and T. Roska, "High performance processor array for image processing", ISCAS 2007, pp. 1177-1180, 2007;

[8] A. Lopich and P. Dudek, "Cellular processor array design in 3D integrated circuit technology", Workshop on 3D integration, Design, Automation and Test in Europe, March 2011;

[9] W. Zhang, Q. Fu and N. Wu, "A programmable vision chip based on multiple levels of parallel processors", IEEE Journal of Solid-State Circuits, Volume 46, Issue 9, pp. 2132-2147, 2011;

[10] V. Krinsky, V. Biktashev and N. Efimov, "Autowaves principles for parallel image processing", Physica D: Nonlinear Phenomena, Volume 49, Issues 1-2, pp. 247-253, 1991;

[11] D. Barr and P. Dudek, "APRON: a cellular processor array simulation and hardware design tool", EURASIP Journal on Advances in Signal Processing, Volume 2009, Article ID 751687, 9 pages;

[12] Tinku Acharya and Ajoy K. Ray, "Image Processing Principles and Applications", Published by John Wiley & Sons, Inc., Hokoken, New Jersey, pp. 135-136, 2005;