Parallelization of Motion JPEG Decoder on TILE64 Many-Core Platform

Xuan-Yi Lin[1], Chung-Yu Huang[1], Pei-Man Yang[2], Tai-Wen Lung[1], Shau-Yin Tseng[2], and Yeh-Ching Chung[1]

[1] Department of Computer Science, National Tsing Hua University
Hsinchu, Taiwan 30013, R.O.C.
{xylin,ychung}@cs.nthu.edu.tw
[2] Information & Communications Research Laboratories
Industrial Technology Research Institute
Hsinchu, Taiwan 310, R.O.C.
{peimanyang,tseng}@itri.org.tw

Abstract. The ubiquity of many-core architectures poses challenges to software developers to make scalable software. To parallelize data-intensive applications on a many-core platform, one has to consider both hardware architecture and software characteristics when writing parallel codes. In this paper, we take Motion JPEG decoder as an example data-intensive application and take TILE64 as an example many-core platform. We parallelize the decoder with two different strategies and observe their impact on program performance and scalability. We design two algorithms, *READ* and *WRITE*, which differ in the direction of data movement between processor cores. Experimental results show that *READ* algorithm outperforms *WRITE* algorithm by 217% when decoding 1080P video on the TILE64 platform. It indicates that the arrangement of data flows in a data-intensive parallel program can have huge impact on program performance and scalability on a many-core platform.

Keywords: many-core architecture, parallel processing, Motion JPEG.

1 Introduction

With rapid industry development of many-core architectures, mass-produced processors now contain tens to hundreds of cores in a single chip. While the trend of processor making is to increase core count rather than processor frequency, software developers can no longer rely on the so called "free lunch" [1] that automatically makes their program run faster on processors clocked at higher frequencies.

For application developers, in order to make the performance of their programs scale well with the number of available cores on many-core architectures, existing software needs to be modified or re-written from ground up. The effort required to adapt existing software to a new many-core processor is directly correlated with the programming language and programming model used. Well understanding of both hardware architecture and software characteristics is also crucial to build scalable software on a many-core platform.

When in the course of parallelize a data-intensive application for a many-core platform, data flow should be considered with hardware architecture in mind. Arrangement of the flow of workloads among processors will have direct impact on the performance and scalability of the adapted program.

In this paper, we explore the method of parallelizing a data-intensive application on a many-core system and observe its impact on program performance and scalability. We take Motion JPEG decoder as an example data-intensive application and TILE64 as an example many-core system. We designed two shared-memory based algorithms, *WRITE* and *READ*, to parallelize a Motion JPEG decoder on the TILE64 platform. *WRITE* is a straightforward algorithm and is easier to implement compared to *READ*. We apply both *WRITE* and *READ* algorithms to an open-source Motion JPEG decoder to evaluate their performance. Benchmark result shows that although the *READ* algorithm requires extra effort and time to implement, it scales far better than the *WRITE* algorithm. The decoder runs as much as 3.17 times faster when adopting the *READ* algorithm instead of the *WRITE* algorithm.

This paper is organized as follows. Section 2 provides background knowledge for TILE64 processor and Motion JPEG files. The *WRITE* and *READ* algorithms are introduced in Section 3 and benchmarked in Section 4. Conclusions of this work are given in Section 5.

2 Preliminaries

2.1 The TILE64 Processor

TILE64 is a general purpose many-core processor made by Tilera [2]. It has an array of 64 identical processor cores (each referred to as a *tile*) interconnected via on-chip two-dimensional mesh networks [tile ref]. TILE64 is fully programmable using standard ANSI C under Linux environment. In addition to standard Linux C, TILE64 can also be programmed using proprietary API called iLib. The iLib library supports two communication mechanisms, shared memory and streaming, for processes running on different cores to communicate. Software developer can use both communication primitives in a program.

Fig. 1 illustrates the architecture overview of a TILE64 processor. There are four memory controllers located at the four corners of processor array. These on-chip memory controllers provide access to an external memory system that is accessible by all tiles. The interface to the memory networks provides access to other tiles and to the DDR2 memory.

To use shared memory mechanisms in a program, the process which is sharing information can call *malloc_shard()* function of the iLib to get an address pointing to a block of shared memory. Then the sharing process notifies other processes the location of shared memory by sending them the pointer to shared memory.

2.2 Motion JPEG

A Motion JPEG (M-JPEG) file is basically a large file containing a sequence of independent JPEG frames. Fig. 2 shows structure of a typical M-JPEG file. There is

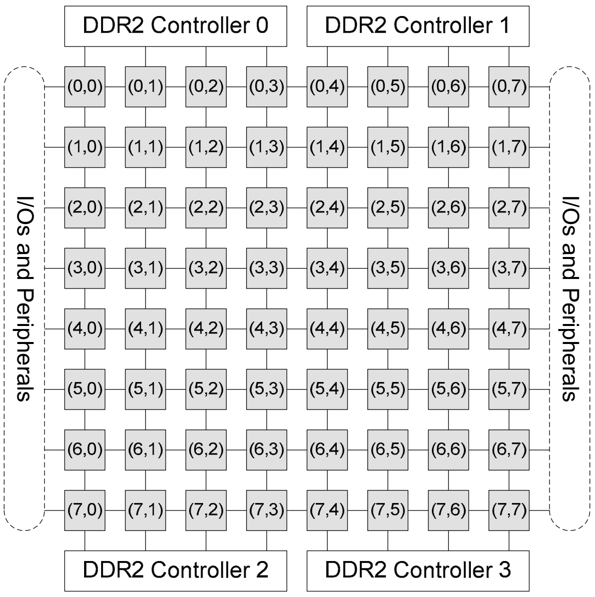

Fig. 1. TILE64 processor architecture overview

no data dependence between frames within an M-JPEG file, thus it is inherently parallel at inter-frame level. The inherent parallelism of an M-JPEG file makes it easy to parallelize an M-JPEG decoder by instructing processors to decode different frames concurrently.

Data size of JPEG frames in an M-JPEG file will vary based on the complexity of individual frames. Decoded YUV frames, however, are equally sized. Fig. 3 illustrates decoding of an M-JPEG file. Because JPEG has high compression rate, size of decoded YUV data is significantly larger than original JPEG data.

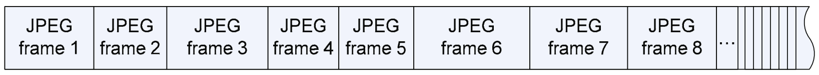

Fig. 2. A Motion JPEG file

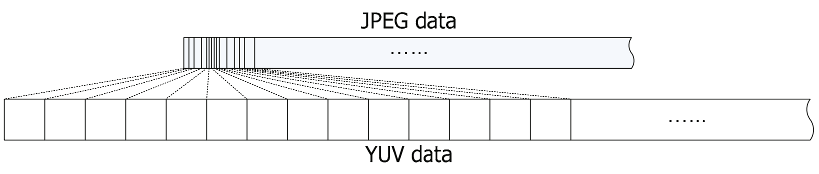

Fig. 3. Decoding of an M-JPEG file into YUV video sequence

3 Parallelization of Motion JPEG Decoder

We design two algorithms, *WRITE* and *READ*, to parallelize Motion JPEG decoder on the TILE64 platform. Both algorithms are shared memory based. JPEG data frames and YUV data frames are moved between tiles using shared-memory mechanism.

In the parallel M-JPEG decoder, there are two process roles, master process and worker process. Master process is responsible for input and output operations. Worker processes are responsible for decoding individual JPEG frames.

Fig. 4 shows a particular instance of processor configuration for both algorithms. In Fig. 4, 32 tiles are working together to decode a M-JPEG file, among the 32 tiles, *tile (0, 0)* acts as master and other 31 tiles serve as workers.

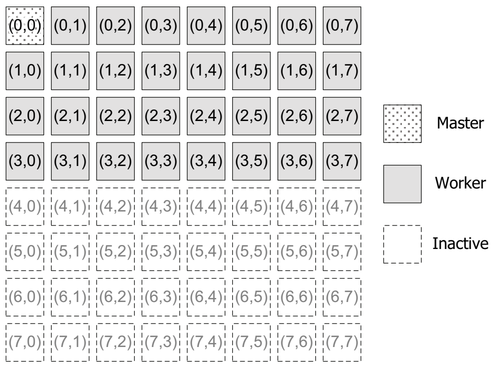

Fig. 4. Decoder configuration

3.1 The *WRITE* Algorithm

Following is the program pseudo code of *WRITE* algorithm for master and worker process. The *WRITE* is a straightforward algorithm. Illustration of the algorithm is given in Fig. 5.

Master process:

```
1.  Initialize shared memory space, JPEG_buffer[] and YUV_buffer[].
2.  Broadcast address pointers of JPEG_buffer[] and YUV_buffer[] to all worker
    processes.
3.  Open and parse input M-JPEG file, mjpegFile.
4.  output_frame_num = 0;
5.  while( frames_to_decode != 0 )
6.  {
7.    if ( JPEG_buffer[] is not full )
8.    Fetch next JPEG frame in mjpegFile and enqueue it to JPEG_buffer[].
9.    if ( YUV_buffer[] is not empty )
10.   {
11.   if ( YUV_frame(output_frame_num + 1) is available and valid )
12.     {
13.     Output( YUV_frame(output_frame_num + 1) );
14.     output_frame_num++;
15.     frames_to_decode – –;
16.     }
17.   }
18. }
```

Worker process:

```
1.  Receive address pointers of JPEG_buffer[] and YUV_buffer[] from master
    process.
2.  while( frames_to_decode != 0 )
3.  {
4.    if ( JPEG_buffer[] is not empty )
5.    {
6.    Move first JPEG frame in JPEG_buffer[] to private JPEG buffer.
7.    private_YUV_buffer = DecodeJPEGframe (private_jpeg_buffer);
8.    Copy private_YUV_buffer to corresponding position in YUV_buffer[].
9.    Set the validity of the YUV frame to valid.
10.   }
11. }
```

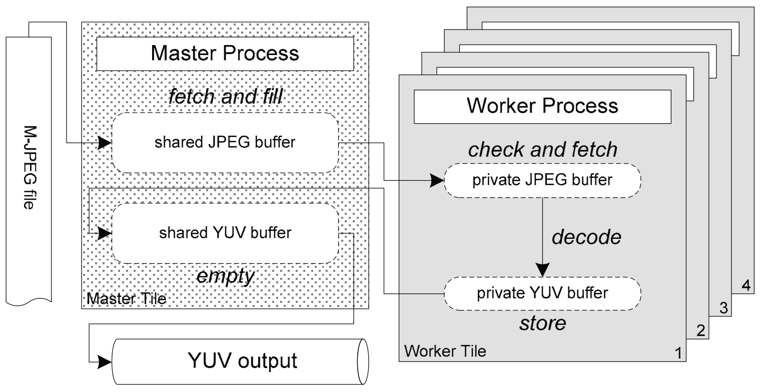

Fig. 5. Illustration of the *WRITE* algorithm

3.2 The *READ* Algorithm

In the *READ* algorithm, every worker process allocates YUV buffer as shared, so the YUV buffer is accessible by master process. Illustration of the algorithm is given in Fig. 6.

Master process:

```
1.  Initialize shared memory space, JPEG_buffer[].
2.  Broadcast address pointers of JPEG_buffer[] to all worker processes.
```

```
3.  Receive address pointers of shared_YUV_buffer from all worker processes.
4.  Open and parse input M-JPEG file, mjpegFile.
5.  output_frame_num = 0;
6.  while( frames_to_decode != 0 )
7.  {
8.      if ( JPEG_buffer[] is not full )
9.          Fetch next JPEG frame in mjpegFile and enqueue it to JPEG_buffer[].
10.     if ( received notification from worker process )
11.     {
12.         Fetch YUV_frame(output_frame_num + 1) from the worker process.
13.         Output( YUV_frame(output_frame_num + 1) );
14.         output_frame_num++;
15.         frames_to_decode – –;
16.     }
17. }
```

Worker process:

```
1.  Receive address pointers of JPEG_buffer[] and YUV_buffer[] from master
    process.
2.  Initialize shared memory space, shared_YUV_buffer.
3.  Send address pointer of shared_YUV_buffer to master process.
4.  while( frames_to_decode != 0 )
5.  {
6.      if ( JPEG_buffer[] is not empty )
7.      {
8.          Move first JPEG frame in JPEG_buffer[] to private JPEG buffer.
9.          shared_YUV_buffer = DecodeJPEGframe (private_jpeg_buffer);
10.         Notify master process the availability of private_YUV_buffer.
11.     }
12. }
```

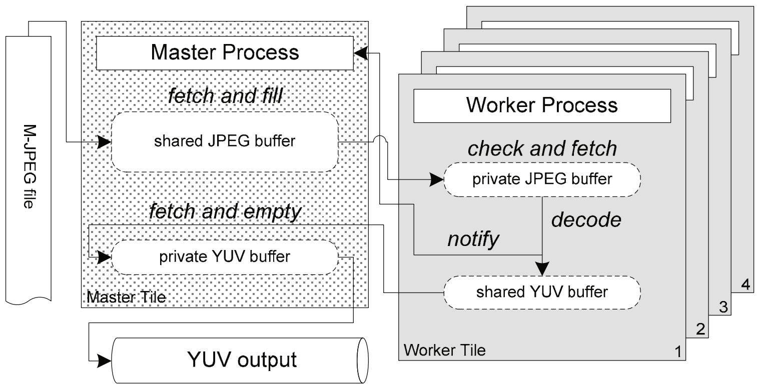

Fig. 6. Illustration of the READ algorithm

4 Experimental Results

We apply *WRITE* and *READ* algorithms to an open source Motion JPEG decoder, *MJPEG Tools* [3] and run the parallelized M-JPEG decoder on TILE64 platform to observe performance and scalability of the decoder. We use the parallel decoder to decode four videos of different resolution. Table 1 lists the test files used.

Table 1. Motion JPEG test files used

	deadline	*city*	*stockholm*	*factory*
Format	CIF	4CIF	720P	1080P
Resolution	352x288	704x576	1280x720	1920x1088
Frames	1374	600	604	1339

4.1 Performance of *WRITE*

Fig. 7 shows speedup of parallel M-JPEG decoder with *WRITE* algorithm using different number of tiles. Number of tiles used shown in the figure, for example 1+15, represents one master process and 15 worker processes.

From the results we can see that the performance of *WRITE* algorithm does not scale beyond 1+15 tiles. To better understand the scalability problem, we also record throughput information of individual tiles and present it visually in Fig. 8 and Fig. 9. Fig. 8 and Fig. 9 show per-tile decoding throughput with master process running on *tile (0, 0)* and *tile (3, 3)* respectively. From Fig 8 and Fig. 9 we can see that worker processes with physical location closer to master process have higher performance. That is because it takes a lot more time for further tiles to write data to the master tile.

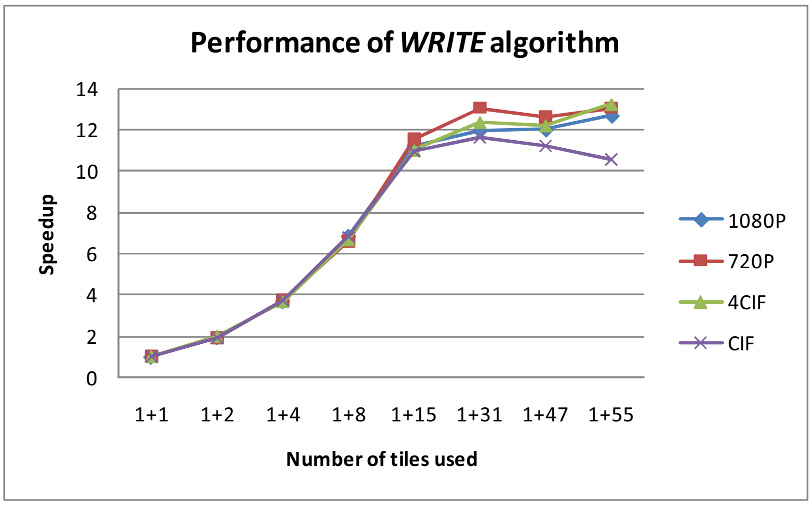

Fig. 7. Decoding performance of parallel M-JPEG decoder using WRITE algorithm

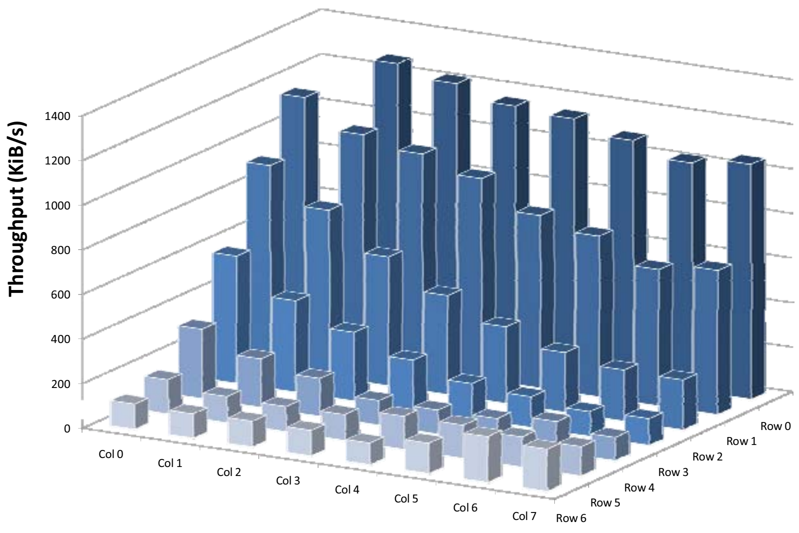

Fig. 8. *WRITE* algorithm per-tile decoding throughput under 1080P workload

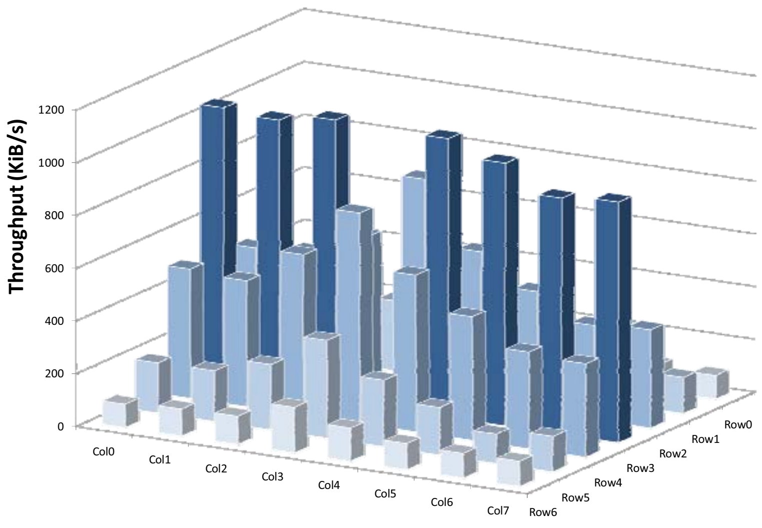

Fig. 9. *WRITE* algorithm per-tile decoding throughput under 1080P workload with master process running on *tile* (3,3)

4.2 Performance of *READ*

Performance of *READ* algorithm is shown in Fig. 10 and Fig. 11. It shows that *READ* algorithm scales beyond 1+31 tiles when decoding a 1080P video file. Throughput data shows that latency of read operation is barely affected by distance between tiles.

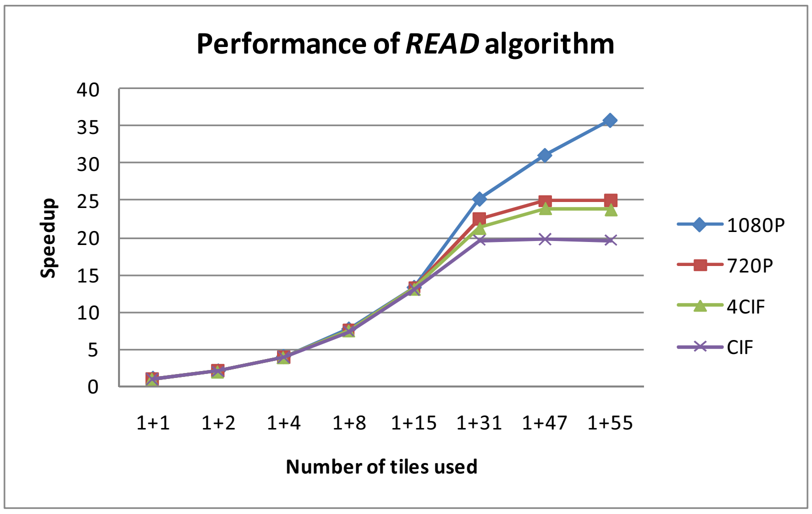

Fig. 10. Decoding performance of parallel M-JPEG decoder using *READ* algorithm

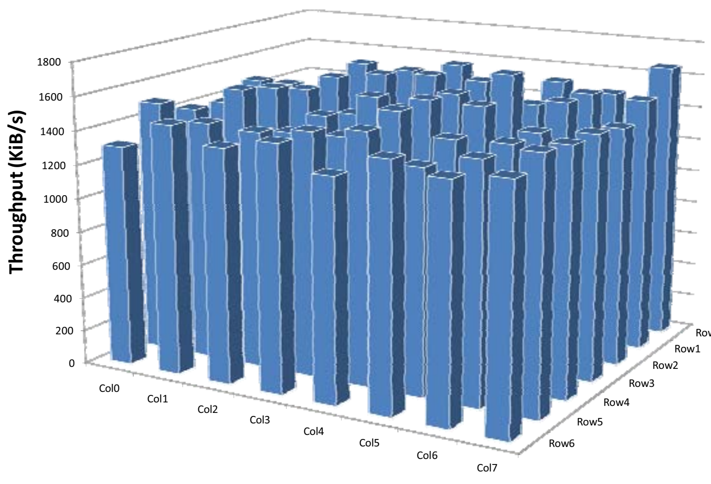

Fig. 11. *READ* algorithm per-tile decoding throughput under 1080P workload

4.3 Performance Advantage of *READ* over *WRITE*

Fig. 12 shows the performance advantage of *READ* algorithm over *WRITE* algorithm. The greatest performance gain can be observed at the configuration of using 1+55 tiles to decode a 1080P video file. It has a performance improvement of 217%. It means that on the TILE64 platform, M-JPEG decoder using *READ* algorithm runs 3.17 times faster than using *WRITE* algorithm when decoding a 1080P video file.

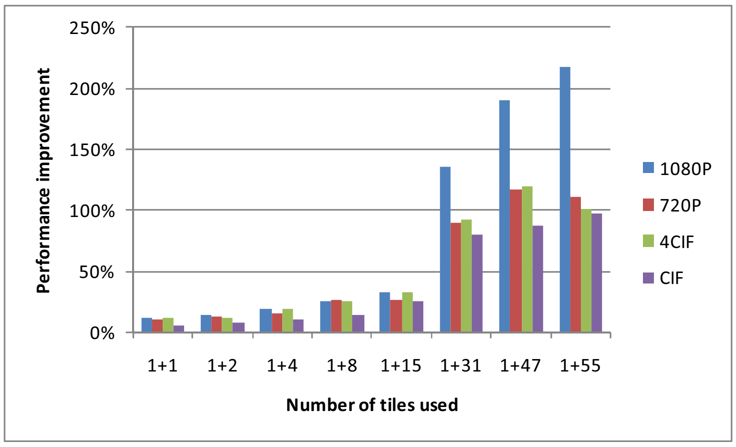

Fig. 12. Performance improvement of *READ* algorithm over *WRITE* algorithm

5 Conclusion

In this paper, we conduct parallelization of Motion JPEG decoder on the TILE64 platform. We want to know how parallelization strategies can impact scalability and performance on data-intensive applications. We designed two share-memory based algorithms, *WRITE* and *READ* to parallelize a Motion JPEG decoder. From the experimental results we have the following remarks:

Remark 1. Parallelization strategy with consideration of both hardware and software characteristics is necessary in building high performance and scalable software on many-core platforms.

Remark 2. On TILE64, latency of write operations to shared memory addresses increases with the distance between sharing tile and writing tile. Read operations are not affected by such overhead.

Remark 3. Although the READ algorithm requires extra implementation overhead, it scales far better than that of the WRITE algorithm.

References

1. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (2005)
2. Tilera Corporation, `http://www.tilera.com`
3. MJPEG Tools, `http://mjpeg.sourceforge.net`