Low-Cost and Low-Power Video Filtering with Parallel Many Cores

Y. Serhan Gener[1], Abdullah Yildiz[1], Sezer Goren[1]

[1]Dept. of Computer Engineering, Yeditepe University, Istanbul
sgener@cse.yeditepe.edu.tr, ayildiz@cse.yeditepe.edu.tr, sgoren@cse.yeditepe.edu.tr

Abstract

This report examines the advantages of low-cost many-core architectures over graphical processing units (GPU) for massively parallel applications. We show that graphical processing units (GPU) could be costly and power inefficient for workloads such as spatial domain video filtering tasks that take advantage of parallel computing. In this study, we analyze a simple video filtering application on a low-cost parallel programming platform called "Parallella" which includes a Xilinx Zynq all programmable system-on-chip (SoC) along with a parallel processing architecture named "Epiphany" co-processor which includes 16 small RISC cores. We compare it to other hardware and software alternatives in terms of cost, power, and performance. We show that it is possible to use such a low-cost platform for these kinds of workloads without too much programming effort.

1. Introduction

Parallel computing is one of the most promising topics nowadays both in hardware and software domains. Various types of workloads could be split into manageable units by executing same or similar operations at the same time.

To take advantages of parallelism in general-purpose hardware architectures, single instruction-multiple data (SIMD) concept has been used for a long time. However, SIMD mechanism on general-purpose processors (GPP) could require the workload to be analyzed during compile-time before running it to better utilize SIMD instructions. Since general-purpose architectures are designed to handle mixed-type of workloads, executing SIMD type of jobs on these kinds of architectures could prevent achieving and exploiting the potential parallelism. For this reason, it is better to use a dedicated hardware to run SIMD type of jobs.

Today heterogeneous architectures can handle various types of workloads by using different kinds of processing units. For example, a heterogeneous computing system can employ a general-purpose processor that takes care of serial programs or low-priority tasks, a co-processor that can run programs to offload compute-intensive tasks from general-purpose processor and also there could be another dedicated processor to handle real-time and critical tasks. However, there are some challenges that should be addressed for a heterogeneous computing platform. Due to the non-uniform architecture of this kind of platform, the management and the interaction between dissimilar processing units has to be handled efficiently.

This work was supported by the Parallella University Program.

Reconfigurable architectures or field programmable gate arrays (FPGA) are well-suited for handling the management and the interaction between dissimilar cores. That is, the field programmable nature of an FPGA could provide on-the-fly update and upgrade to handle the interconnection between different types of computing units. A good example to this concept is Parallella [1], which includes a dual-core ARM A9 processor along with an FPGA and a 16-core co-processor.

In this report, we analyze this platform and try to take advantage of this heterogeneous architecture on a simple video filtering application with respect to competitive GPUs.

2. Background

Image and video processing are one of the areas of interest in parallel computing and as a platform GPUs are mostly preferred for their tightly coupled software and hardware frameworks for these types of workloads. Compared to a general purpose processor (GPP) including a few cores optimized for sequential processing, a GPU has a massively parallel architecture consisting of many smaller cores designed for handling and processing multiple tasks simultaneously. Many application domains requiring high-performance computing could also take advantage of such a platform. However, since GPUs are usually power-hungry devices, using them on power-sensitive embedded platforms could be costly.

Since we're targeting low-cost and low-power implementation of image and video processing tasks, it makes sense to analyze some low-end commercial GPUs. Table 1 shows GPU models supporting 'Compute Unified Device Architecture (CUDA)' of NVIDIA [2] in terms of memory bandwidth, performance, and power.

Table 1. Technical specs for low-end and low-power GPUs supporting CUDA

GPU Model	Memory Bandwidth (GB/s)	Processing Power (GFLOPS)	Power Consumption (Watts)	GFLOPS/W
GeForce 710M	14.4	307.2	12	25.6
GeForce 820M	16	366.3	15	24.42
GeForce GT 420M	25.6	192	10	19.2
GeForce 610M	14.4	142.08	12	11.84
GeForce GT 220M	25.6	120	14	8.57
GeForce GT 320M	25.3	90	14	6.43
GeForce G 102M	6.4	48	14	3.43
GeForce 8400M G	6.4	19.2	10	1.92

As shown in Table 1, low-end GPUs have considerable power requirements to exploit parallelism in a low-cost system. When we think of applications that could run on embedded platforms, power becomes the most important aspect of the system. On the

other hand, on an embedded platform, the overhead of transferring workloads from memory to GPU could become challenging as interaction with GPU is usually done over PCI interface. However, using a reconfigurable interface could make this data exchange more flexible and power efficient. For this reason, using FPGAs would be convenient in this sense.

3. Parallella as a Parallel Computing Platform

Parallella is an energy-efficient many-core platform which includes 16-core Adapteva Epiphany co-processor and Xilinx Zynq-Z7010 all programmable SoC. Besides its potential as a heterogeneous computing platform alone, its small size also makes it an ideal component for parallel computing applications running in a cluster configuration. Figure 1 shows the top-view of Parallella.

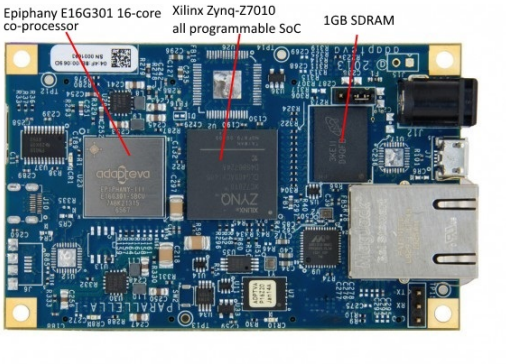

Figure 1. Parallella platform

The Epiphany co-processor on Parallella platform consists of a scalable array of 16 small RISC processors that are programmable in bare metal C/C++ or in a parallel programming framework like OpenCL, MPI, and OpenMP. These mesh of independent cores are connected together with a fast on-chip network within a distributed shared memory architecture. Figure 2 shows the block diagram of the Epiphany 16-core co-processor.

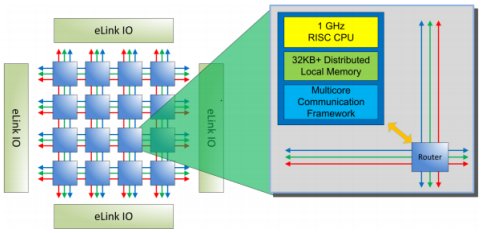

Figure 2. Epiphany E16G301 16-core block diagram

The Epiphany co-processor is connected to the Zynq SOC via the *eLink* interface. An Epiphany *eLink* protocol stack is implemented in the reconfigurable logic portion of the Zynq SOC. In addition to the *eLink* interface, the programmable logic portion also includes an AXI master interface, an AXI slave interface, and an optional HDMI controller interface within the reference design by default.

Xilinx Zynq-Z7010 all programmable SoC is the host device on Parallella and runs a Linux operating system. It includes a dual-core ARM A9 CPU along with programmable logic that is equivalent to 430K ASIC (Application Specific Integrated Circuit) gates in size. The existence of such a programmable logic makes the interface between the co-processor and the host CPU flexible and reconfigurable. Figure 3 shows the high level architecture of Parallella.

In terms of power consumption, the Epiphany co-processor has some advantages against low-end GPUs. It has less than 2W power consumption while achieving 32 GFLOPs peak performance. This corresponds to 16 GFLOPS/W such that this performance/power ratio is comparable to CUDA supported low-end GPUs with respect to Table 1. The off-chip memory bandwidth of the Epiphany co-processor is 8 GB/s.

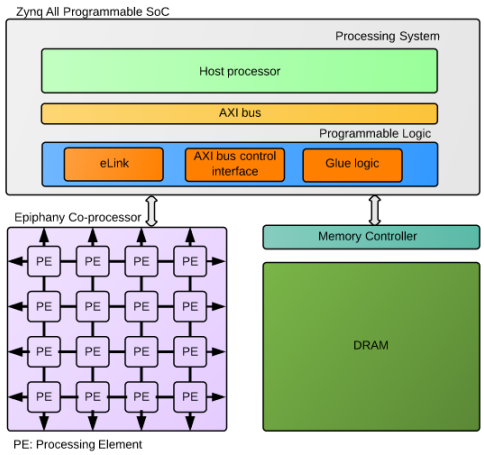

Figure 3. Parallella high level architecture

4. Spatial Domain Image and Video Filtering on Parallella

We implemented a simple video processing application on Parallella platform in order to show that it has comparable performance/power ratio with respect to low-end GPUs. An input video feed was obtained from Kinect Xbox 360 device which has 640x480 pixels color resolution although any camera would do the job. The reason that we selected Kinect rather than a regular camera is also to use it to process depth information in a future work.

In this implementation, each core on the Epiphany co-processor processes a segment of the current video frame by applying a median filter and then an edge detection function using Laplacian kernel. Each core uses its own local memory to read the data and write the result as well as for the synchronization with the host. Figure 4 shows the sequence diagram of our implementation.

To implement the software running on the host, we used *libfreenect* library to take the RGB video feed of Kinect over USB interface into the host part of Parallella platform and then we transformed it into grayscale color space. After that, the transformed video feed was transferred into the local memories of the Epiphany co-processor's cores in order to perform the filtering operation. To accommodate the interaction between the host and the Epiphany co-processor, we used *Epiphany Hardware Abstraction Layer (eHAL)* library and to manage and con-

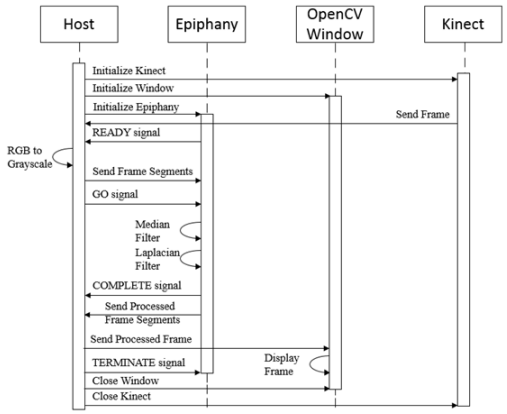

Figure 4. Sequence diagram of video filtering application we implemented

figure the Epiphany co-processor, we used *Epiphany Hardware Utility Library (eLib)*.

4.1. Implementation of the host side

The ARM processing system on the Zynq SoC (i.e., the host) is responsible for the initialization of the Epiphany co-processor and Kinect. After initializing them, the host waits for every core on the Epiphany co-processor to return a *READY* signal which indicates that they are ready to receive data and process it. When the *READY* signal is sent from each core, an RGB video frame is received from Kinect and then converted to grayscale colorspace on the host. This grayscale video frame is then partitioned to be shared among the cores of the Epiphany co-processor and written to each core's local memory. After that a *GO* signal is set on each core's local memory to start them to process data. That is, the *GO* signal indicates that a new video frame has been uploaded to the core's local memory and waits for being processed. Then the host receives the next RGB video frame from Kinect, converts it to grayscale colorspace as each core processes their corresponding segment of the video frame and waits for a *COMPLETE* signal from each core to acknowledge that the corresponding core completed its processing and is ready to receive the part of the next video frame. After all the cores complete their processing, the processed frame segment is read from each core's local memory and combined on the host to reconstruct the processed video frame. The reconstructed frame is displayed via the host and the flow starts again by writing the next frame to cores' local memories. Pseudo code relating to the implementation of the host side is shown in Listing 1.

4.2. Implementation of the Epiphany Co-processor side

From the perspective of the Epiphany co-processor, each core on it starts their execution after the host loads the program code into their local memory. The Epiphany co-processor we used consists of sixteen cores and each core runs the same program. The pseudo code of the program is shown in Listing 2. When the Epiphany system is loaded, cores first read their 12-bit core IDs from the core registers. With the core ID, each core calculates the row and column coordinates in its own workgroup[1] and also calculates their respective core number. Before cores became ready for the processing, each core calculates the number of rows they are going to receive from the video frame and then sets the *READY* signal in their local memory so that the host can acknowledge that they have completed their initialization and been loaded successfully. Then every core waits for the *GO* signal to be set by the host to indicate that the host has just written a new frame section to their local memory. After a core finishes its processing, it sets the *COMPLETE* signal on its local memory for the host to read, so that the host can acknowledge the completion of the processing at the corresponding core. After setting the *COMPLETE* signal, each core also resets the *GO* signal which was set by the host so that they start waiting for taking the segment of the next video frame from the host to process.

Listing 1: Pseudocode of the host part

```
INITIALIZEKINECT()
E_INIT(NULL)
E_RESET_SYSTEM()
E_GET_PLATFORM_INFO(&e.platform)
msg.signal_terminate=0
msg.signal_go=0
msg.core_ready=0
msg.core_complete=0
for row := 0 to e.platform.rows do
    for col := 0 to e.platform.cols do
        E_WRITE(&e.epiphany, row, col,
        MESSAGE_ADDRESS, (void *)(&msg),
        sizeof(msg))
    end
end
WAITCOREREADY()
INITIALIZEWINDOW()
RGBTOGRAY( GETFRAME() )
while true do
    WRITEFRAMETOCORES()
    SIGNALGO()                    ▷ send Go signal to every core
    RGBTOGRAY( GETFRAME() )
    WAITCORECOMPLETE()    ▷ wait every core to complete
    computation
    READFRAMEFROMCORES()  ▷ read frame segments from
    every core
    SHOWIMAGE()
    c = WAITKEY(33)                          ▷ ESC Pressed
    if c == 27 then
        SIGNALTERMINATE()      ▷ send Terminate signal to
        every core
        break
    end
end
RELEASESOURCES()
```

4.3. Segmentation of the video frame

After converting the video frame[2] from RGB to grayscale color space, the host starts to write frame segments to each Epiphany

[1] A workgroup is a collection of adjacent cores on Epiphany co-processor(s).

[2] The resolution of the video frame is 640x480 pixels.

Listing 2: Pseudocode of the Epiphany co-processor part

```
coreid = E_GET_COREID()
ECOORDS_FROM_COREID(coreid, row, col)
cnum = row * e_group_config.group_cols + col
core_ready = 1
while signal_terminate != 1 do
    while signal_go != 1 do
    end
    MEDIANFILTER(cnum)
    EDGEDETECTIONFILTER()
    core_complete = 1
    signal_go = 0
end
```

co-processor core. The size of a frame segment is shown in Figure 5. Segments are sent to the Epiphany co-processor cores with overlapping pixel regions to perform the filtering easily. Every core receives segments with a resolution of 32x640 pixels where the first row of a segment overlaps with the last row of the previous segment and the last row of a segment overlaps with the first row of the next segment, except the first and the last cores. The first and the last cores receive segments with a resolution of 31x640 pixels. Overlapped segmentation is shown in Figure 6.

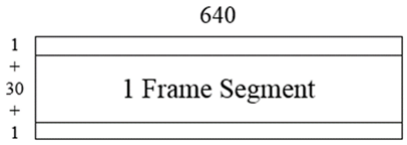

Figure 5. Frame segment that is processed by each core on the Epiphany co-processor

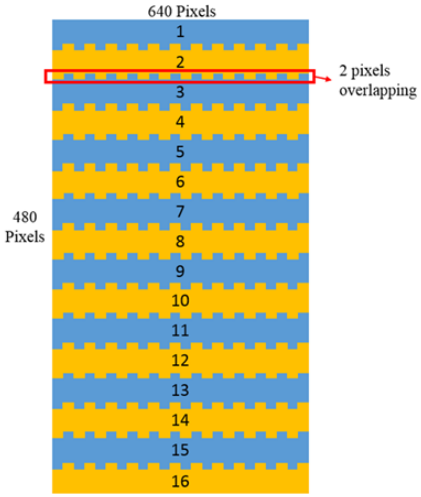

Figure 6. Overlapping segmentation of a video frame

5. Test & Results

We implemented the same filtering algorithms without using any library implementations of them both on Parallella and GPU platforms. To compare with Parallella, we used NVIDIA GT 520 GPU which has 155.5 GFLOPS peak performance and consumes 29 Watts of power.

For a fair comparison, we utilized the same amount of cores on both platforms. We measured latencies of each particular operation on both platforms and then calculated frame per second (FPS) rate to show that how good they exploit parallelism. The results of this comparison are shown in Table 2.

Table 2. Comparison of the Epiphany co-processor with a low-end GPU[3]

Accelerator Used	Number of Cores Utilized	Frame Rate (FPS)	Latency	
			Data Transfer between Host and Accelerator	Median Filtering & Edge Detection
Epiphany E16G301	16/16	4.12	38ms	131ms
NVIDIA GT 520[4]	16/48	2.97	1ms	323ms

We observed that the latency caused by the data transfer on Parallella platform is between 30ms and 50ms. The actual data processing needed 131ms on average.

As a result, we can say that the Epiphany co-processor is an ideal low-power hardware accelerator with respect to an equivalent GPU platform.

6. Conclusion

In conclusion, it was shown that Parallella platform can be used to accelerate applications which inherently supports parallelism.

As a feature work, our implementation can be improved by changing the memory transfer structure between DRAM and the Epiphany co-processor. In this way, it is possible to use more than one matrix or a larger matrix for processing data on the Epiphany co-processor. For example, one additional matrix can be the depth information which can be acquired from Kinect or larger matrix frames received from a camera can be used directly to obtain a more visual representation.

As an another example, depth information can be used to get the position of the nearest object in a video frame. Later on, this information can be used to remove the background and the objects which are located at further back. For example, it can be used to extract information from a camera at an ATM. Hence, with the provided implementation it can be possible to eliminate people at the back that are only passing through and only focus on the person that is in front of the ATM.

Other than image processing applications, Parallella platform can also be used for artificial intelligence (AI) applications. In general, AI applications require a lot of computational power when building tree structures and searching in them with a massively parallel way. Since Parallella is physically small and power efficient, it could be an ideal computing platform in a robotic device which runs AI algorithms.

[3]The results are rough estimates.

[4]GPU operates on a PC which has Intel Core2 Duo CPU E7300 2.66GHz along with a 4GB DRAM.

7. References

[1] Supercomputing for Everyone
`http://www.parallella.org/`

[2] NVIDIA
`http://www.nvidia.com/`

8. Appendices

A.

Listing 3: Pseudocode of the median filter function

```
for row := 1 to 30 do
    for col := 0 to 639 do
        for wrow := −1 to 1 do
            for wcol := −1 to 1 do
                window[wrow+1][wcol+1] =
                frame[row+(wrow+1)][col+(wcol+1)]
            end
        end

        sort(window)

        if row == 0 then
            tmp[col] = window[4]
        end
        else
            frame[row-2][col] = tmp[col]
            tmp[col] = window[4]
        end
    end
end

for col := 0 to 639 do
    frame[29][col] = tmp[col]
end
```

B.

Listing 4: Pseudocode of the edge detection function

```
window[9] = {-1,-1,-1,-1,8,-1,-1,-1,-1}

for row := 0 to 29 do
    for col := 0 to 639 do
        for wrow := −1 to 1 do
            for wcol := −1 to 1 do
                value = window[wrow+1][wcol+1] *
                frame[row+wrow][col+wcol]
            end
        end
    end
end

if row == 0 then
    tmp[col] = (value / 6 < 0 ? 0 : 255)
end
else if col == 0 then
    temp = tmp[col]
    tmp[col] = (value / 6 < 0 ? 0 : 255)
end
else if col == 639 then
    frame[row-1][col-1] = temp
    frame[row][col] = tmp[col]
    tmp[col] = (value / 6 < 0 ? 0 : 255)
end
else
    frame[row-1][col-1] = temp
    temp = tmp[col]
    tmp[col] = (value / 9 < 0 ? 0 : 255)
end

for col := 0 to 639 do
    frame[29][col] = tmp[col]
end
```