GPU-EvR: Run-time Event Based Real-time Scheduling Framework on GPGPU Platform

Haeseung Lee, Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science
University of California Irvine, Irvine, CA, USA
E-mail: {haeseunl, alfaruqu}@uci.edu

***Abstract*—GPU architecture has traditionally been used in graphics application because of its enormous computing capability. Moreover, GPU architecture has also been used for general purpose computing in these days. Most of the current scheduling frameworks that are developed to handle GPGPU workload operate sequentially. This is problematic since this sequential approach may not be scalable for real-time systems, which is a consequence of the approach's inability to support preemption. We propose a novel scheduling framework that provides real-time support for the GPGPU platform. In contrast to existing frameworks, our proposed framework considers both concurrent execution of applications on the GPU and mapping between streaming multiprocessors and thread blocks. By considering both concurrent execution and mapping, our framework is able to guarantee timing up to 6.4 times as many applications compared to TimeGraph [9] and Global EDF [5]. In addition, our experimental applications use up to 20% less power under our scheduling framework compared to [5], [9].**

I. Introduction and Related Work

The demand for powerful processing element derives the advance of hardware architecture. Numerous multi-core architectures are developed to comply with the demand. One of the most powerful architectures is GPU. GPU architectures have multiple compute intensive processors which are specialized to perform SIMT (*Single Instruction Multiple Thread*) operations [14]. The performance of this powerful processing unit is at least six times faster compared to a general purpose CPU architecture [17]. In order to maximize the utilization of a GPU, a platform which is called general-purpose computing on graphics processing units (GPGPU) has emerged [16]. In general, GPGPU programs are composed of two parts: host code and kernel code [14]. Host code is executed on the CPU and mainly includes data transfer between the host and the GPU, the preparation process for the GPU device, and launches kernels with configuration. On the other hand, kernel code is executed on multiple GPU cores with the configuration.

In order to use the GPUs in real-time systems, research has been conducted. Elliott et al. [7] discussed the types of applications that may use GPUs in real-time systems and the limitation of GPU support for real-time systems. Liang et al. [10] addressed GPU based real-time implementation of 3D sound localization platform. Kato et al. [8] proposed a technique to reduce the delay which is caused by memory transfer between the I/O device and GPUs. The proposed technique creates the direct mapping between the I/O device and GPUs to reduce memory transfer delay. Mu et al. [12] proposed GPU implementation of *High Performance Embedded Computing* benchmark suites (HPEC) which includes several signal processing applications. Zhu et al. [19] proposed CPU/GPU integrated micro-architecture which improves QoS for IP routing. The suggested micro-architecture improves performance by using GPU for IP packet processing. These works focused on powerful computing capability of the GPUs. GPU performs computational part of the real-time applications to satisfy the timing requirements. These works did not consider the multi-tasking environment that GPU likely needs to handle multiple applications at the same time.

However, there is a problem in the traditional GPGPU programming model to handle multiple applications concurrently. By default, GPU executes kernels sequentially; one kernel at a time. Recent CUDA and NVIDIAs GPU architectures may execute multiple different kernels if there are available resources [13]. If the resources are not enough, the GPU executes kernels sequentially. Sequential kernel execution could provide enough performance in most of the general purpose computing. However, in the real-time domain, sequential execution may cause problems because there is a possibility for priority inversion [9].

Many techniques have been developed to overcome this problem. Elliott et al. [6] proposed a locking protocol for globally-scheduled *Job-Level Static-Priority* (JLSP) real-time systems which use GPUs as shared resources. Ward et al. [18] proposed priority donation based locking protocols for globally-scheduled multi-processor real-time systems. Membarth et al. [11] proposed a dynamic task-scheduling and resource management mechanism in medical imaging. Basaran et al. [2] addressed a task preemption technique for real-time systems by dividing a large kernel into small sub-kernels. These works uses a locking protocol or divide workload into small piece to support preemption. Kato et al. [9] proposed GPU scheduler which is called *TimeGraph* for periodic workload. *TimeGraph* assigns different scheduling properties to applications based on the nature of the applications. From both the scheduling property and the priority of the application, *TimeGraph* selects the application and submits the application to the GPU. However, *TimeGraph* considers a small number of combinations of applications. Elliott et al. [5] discussed real-time scheduling algorithm for multiple CPU single GPU system. The proposed approach treats the GPU as a shared resource and implements *Global Earliest-Deadline-First* [4] based locking protocol to improve overall system efficiency. But, only periodic applications are considered as the target applications.

The problem of previous research is that concurrent execution of applications on the GPU and mapping be-

978-3-9815370-2-4/DATE14/©2014 EDAA

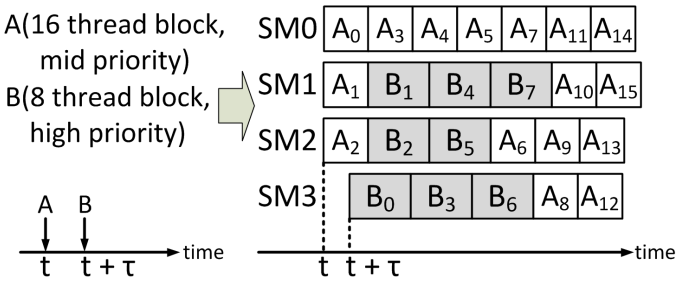

Fig. 1: Scheduling example of our proposed execution model

tween streaming multiprocessors and thread blocks are not considered. In order to solve this problem, we propose a novel scheduling framework (GPU-EvR). Our proposed GPU-EvR framework provides application-specific timing guarantees by using a two-level preemption support.

The rest of this paper is organized as follows: Section II discusses our contributions and problem description. Section III describes our execution and timing model. Section IV explains the proposed GPU-EvR framework. Section V provides experimental results and Section VI concludes the paper.

II. Contributions and Problem Description

The novel contributions of this work are:

- A run-time event based scheduling framework (GPU-EvR) for the GPU-based real-time embedded systems. The execution model of the GPU-EvR creates mapping for concurrent execution of multiple applications and the timing model of GPGPU workload predicts the current status of the applications.
- A two-level GPU resource management algorithm to provide fine-grained GPU resource management to ensure application-specific timing guarantees are met.
- Our proposed GPU-EvR is evaluated by comparing with *TimeGraph* [9] and *Global-EDF* [5] scheduling framework. The number of applications which may meet the timing requirement is up to 6.4 times as many compared to both *TimeGraph* and *Global-EDF* scheduling frameworks. Moreover, GPU-EvR framework works well with constrained power budgets, since applications running through GPU-EvR consume less power than other frameworks.

A. Problem description

We consider a GPU-based real-time embedded systems as our target platform. The GPU has a total P_{tot} streaming multiprocessors. A set of random application $S = \{<A_1, T_1^{req}>, ..., <A_i, T_i^{req}>\}$ is executed by the user and submitted to the GPU. A_i represents the application and T_i^{req} represents the timing requirement. During the system operation, $S' \subset S$ represents the set of applications which is executed on the GPU at the same time. In addition, response time requirement T_i^{resp} is obtained by using T_i^{req} when the application A_i is dispatched by the user. P_i denotes the GPU resources for $A_i \in S'$. $Res(A_i)$ and $E(A_i)$ represent response time and execution time of the A_i, respectively.

For a given profiling data for the application A_i and the response time requirement T_i^{resp}, our scheduling framework generates a schedule that maps the GPU resource to A_i such that $P_{tot} \geq \sum_{\forall j} P_i$ and $Res(A_i) \leq T_i^{resp}$.

III. GPGPU Model for Real-time Systems

For our scheduler framework, we characterize the behavior of the applications and create execution and timing models. We also make several assumptions for the target systems and the applications.

- Applications have at least one compute intensive kernel function. The kernel function handles most of the computational part of the application and is executed on the GPU.
- High and medium priority applications may have both short and long timing requirements. However, low priority applications may have only long timing requirements.
- Because of the limitation of the current GPU execution model, applications may not be suspended after starting their execution on the GPU.

A. Execution Model for GPGPU workload

In the GPGPU programming model, when the GPU executes a kernel function, the mapping between *Streaming Multiprocessors* (*SMs*) and thread blocks is created [14]. A single thread block is processed by a single SM. Our scheduler creates the mapping between thread blocks and SMs based on application priority. When a higher priority application is submitted to a GPU, the scheduler allocates GPU resources following the order of the priority.

Figure 1 gives a simple example of our execution model. In this example, we assume our GPU has four streaming multiprocessors and two applications (A and B) are injected at time t and $(t + \tau)$, respectively. At time t, scheduler allocates three streaming multiprocessors for application A. After time τ, the higher priority application B is submitted to the system. At $(t+\tau)$, only one streaming multiprocessor is available. Therefore, application B starts its operation with a single streaming multiprocessor. The scheduler allocates two more streaming multiprocessors to application B after the first three thread blocks of application A are completed. After completion of application B, application A uses the entire GPU to meet the deadline.

B. Timing Model for GPGPU workload

The timing model aims to provide enough information to the scheduler to obtain the needed amount of GPU resources in order to guarantee the timing. In addition to the variables defined in Section II-A, $R(A_i, P_i)$ depicts a remaining execution time of A_i with GPU resource P_i, $K_{i,k}$ describes the k^{th} kernel of the application A_i, and $N_{i,k}$ represents the number of thread blocks for the kernel $K_{i,k}$. Total execution time of an application $E(A_i)$ is a summation of the individual kernel execution time $K_{i,k}$ and the memory transfer delay (Equation 1).

$$E(A_i) = \sum_{k=0}^{|K_i|-1} E(K_{i,k}) + D_{memcopy} \quad (1)$$

Since target applications are compute intensive applications, we may neglect memory transfer delay $D_{memcopy}$ and therefore the execution time of an individual kernel $E(K_{i,k})$ depends on GPU resource. Since kernel functions may be executed with different amount of resources, kernel execution time depends on currently assigned GPU resource

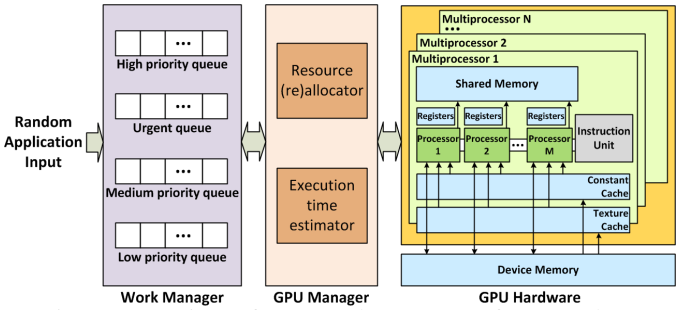

Fig. 2: Overview of proposed GPU-EvR framework

P_i for the application A_i. Thus, execution time of k^{th} kernel $E(K_{i,k})$ can be re-written to $E(K_{i,k}, P_i)$. Therefore, total execution time of an application may be re-written as:

$$E(A_i) = \sum_{k=0}^{|K_i|-1} E(K_{i,k}, P_i) \tag{2}$$

$$T_i^{resp} = T_i^{init} + T_i^{req} \tag{3}$$

In order to meet the timing requirement, the total response time of an application must be smaller than the application-specific timing requirement. Total response time requirement T_i^{resp} is a summation of application dispatch time T_j^{init} and timing requirement T_i^{req} (Equation 3). Therefore, when GPU is not fully occupied, we may write the acceptable condition as:

$$T_i^{resp} \geq Res(A_i) = \sum_{k=0}^{|K_i|-1} E(K_{i,k}, P_i) \tag{4}$$

If the GPU is fully occupied, the total response time of the application also includes the waiting delay. The waiting delay is the minimum value among the remaining execution time of currently executing applications on the GPU. Therefore, updated response time of the application $Res(A_i)$ is:

$$Res(A_i) = min(R_{A_j \in S'}(A_j, P_j)) + \sum_{k=0}^{|K_i|-1} E(K_{i,k}, P_j) \tag{5}$$

The remaining execution time of the application $R(A_j, P_j)$ is the summation of remaining execution time of the currently executing kernel and the execution time of the remaining kernels. Let k' represents the index of the currently executing kernel. We can then describe the remaining execution time of the applications for given GPU resource P_j by using the following equation.

$$R(A_j, P_j) = R(K_{j,k'}, P_j) + \sum_{k=k'+1}^{|K_i|-1} E(K_{j,k}, P_j) \tag{6}$$

The remaining execution time of the kernel is a function of the number of processed thread blocks N' and total number of thread blocks $N_{k'}$. Thus, the remaining execution time for a kernel may be obtained by the following equation.

$$R(K_{j,k'}, P_j) = \frac{N_{j,k'} - N'_{j,k'}}{N_{j,k'}} \times E(K_{j,k'}, P_j) \tag{7}$$

Whenever there is a change in S', the scheduler will reallocate GPU resources if there are higher priority applications which do not have enough GPU resources.

Algorithm 1: Workload manager algorithm of GPU-EvR

```
1  while true do
2      UrgentApp = NULL; // Urgent application
3      NextApp = NULL; // application submitted to
       the GPU
4      AvailSM = 0; // available GPU resource
5      RequiredSM = 0; // required SMs
6      UsedSM = 0; // actually assigned SMs
7      UrgentApp = CheckAndUpdateAppInfo();
8      if UrgentApp != NULL then
9          UrgentQueue.push(UrgentApp);
10     AvailSM = GPU.availSM();
11     if NumWaitingApp() > 0 and AvailSM > 0 then
12         NextApp = SelectNextApp();
13         RequiredSM = GetRequiredResource(NextApp);
14         if AvailSM ≥ RequiredSM then
15             NextApp.HasEnoughRes(true);
16             UsedSM = RequiredSM;
17         else
18             NextApp.HasEnoughRes(false);
19             UsedSM = AvailSM;
20             GPU.SetReallocFlag(true);
21         GPU.SubmitApp(NextApp, UsedSM);
```

$$T_i^{resp} \geq T_{current} + R(A_j, P_j) \tag{8}$$

By using the application start time T_j^{init} and the current time stamp $T_{current}$, our scheduler finds the amount of resources that satisfies Equation 8.

IV. Real-time GPGPU Scheduling Framework

Our scheduler framework consists of two levels: workload manager and GPU manager. Figure 2 describes the overview of our framework. After applications have been initiated by the user, the workload manager decides which application will be submitted to GPU based on their priorities. At run-time, the GPU manager keeps tracking the status of the GPU resources and (re-)allocates GPU resources based on the priority and the timing requirements of the applications.

A. Workload Manager

An application is started with the following information: timing requirement, profiling data, priority, and so on. The application is directly submitted to the GPU if there are no waiting applications and the GPU has available resources. Otherwise, the workload manager classifies the application based on its priority and pushes it into a corresponding queue. Whenever there are available resources in the GPU, the workload manager selects an application from any non-empty waiting queues based on the priority.

However, a starvation problem may occur with this approach. Since a higher priority application is selected, there is a possibility that lower priority applications never use GPU resources. In order to prevent the starvation problem, the workload manager has a special type of application queue which is called the *urgent queue*. During run-time, the workload manager keeps track of the current system time $T_{current}$, and the response time requirement of the application T_i^{resp}.

$$T_i^{margin} = T_i^{resp} - E(A_i) \tag{9}$$

Algorithm 2: GPU manager algorithm of GPU-EvR

```
1  while true do
2      ForceLowerApp = false; // flag to include lower
       priority apps
3      RequiredSM = 0; // reallocatable SMs
4      UsingSM = 0; // actually assigned SMs
       /* resource reallocation variable
          initialization                          */
5      SMForRealloc = false;
6      ReAllocIdx = 0;
7      ReallocList.clear();
8      GPU.ExecuteApp();
9      if GPU.GetReallocFlag() then
10         [ReallocList, SMForRealloc] = CreateReallocList();
11         while ReallocList.size() > 0 and SMForRealloc > 0 do
12             RequiredSM =
               GetNewResource(ReallocList[ReallocIdx]);
13             if SMForRealloc ≥ RequiredSM then
14                 UsingSM = RequiredSM;
                   ReallocList[ReAllocIdx].HasEnoughRes(true);
15                 SMForRealloc = SMForRealloc - RequiredSM;
16             else
17                 UsingSM = SMForRealloc;
                   ReallocList[ReAllocIdx].HasEnoughRes(false);
18                 SMForRealloc = 0;
19             ReallocList[ReAllocIdx].Realloc(UsingSM);
20             ReallocList.pop(); ReAllocIdx++;
21     if GPU.availSM() > 0 then
22         NotifyToWorkManager();
```

By using Equation 9, the workload manager may obtain timing margin of the application T_i^{margin}. Therefore, the application has to start its execution on a GPU before T_i^{margin} in order to meet the timing requirement. The workload manager compares current system time $T_{current}$ and T_i^{margin} of the medium and low priority applications. If current system time is close to T_i^{margin} of the application, then the workload manager classifies the application as an urgent application and pushes the application onto the *urgent queue*.

Algorithm 1 describes the pseudo code of the workload manager of GPU-EvR. After the system starts, variables are initialized in Line 2-6. In Line 7, urgent applications are selected from the waiting queues. If there are urgent applications, the workload manager pushes those applications onto the urgent queue in Line 8-9. Current resource status of the GPU is obtained in Line 10. If there are available GPU resources, the workload manager obtains the application from waiting queues in Line 12 and calculates amount of resources which are required to meet the timing requirement in Line 13. In Line 14-16, if currently available GPU resources are greater than or equal to required amount of resources, then the workload manager assigns required amount of resources to the application. However, in Line 18-19, the application will use currently available resources when the GPU does not have enough resources. After that, the workload manager requests resource reallocation to the GPU manager in Line 20. In Line 21, the workload manager submits the application to the GPU.

The functions $CheckAndUpdateAppInfo()$ has a complexity of $O(|A'|)$. $SelectNextApp()$ has a complexity of $O(4)$ and $GetRequiredResource()$ has a complexity of $O(|P_{tot}|)$. Consequently the overall complexity of the workload manager algorithm is given by $O(|A'| + 4 + |P_{tot}|) = O(n)$.

B. GPU Manager

As mentioned in Section III-A, our scheduler creates and modifies mapping between the streaming multiprocessors and the thread blocks of applications. As the GPU resource is occupied or released by the applications, the status of the GPU resources keeps changing at run-time. Based on our execution model (Section III-A), we can derive the following two cases for resource reallocation:

- **Higher priority applications are not submitted with enough resources**: In this case, resource reallocation is required to allocate enough GPU resources to higher priority applications.
- **The application completes its operation on the GPU**: After the application completes its work, GPU resources are released and made available to other applications. Therefore, currently executing applications are able to use more GPU resources through resource reallocation.

At the first stage of resource reallocation, the GPU manager creates a resource reallocation list and obtains the GPU resources for resource reallocation. While creating the resource reallocation list, the GPU manager checks the resource status of the application in a priority order. If the application has enough GPU resources to meet the timing requirement, the GPU manager keeps the current status. However, if the application does not have enough GPU resources, all the lower priority applications are included on the resource reallocation list. After creating a resource reallocation list, the GPU manager estimates the amount of resources to meet the timing requirement in a priority order with Equation 6.

Algorithm 2 describes the pseudo code of the GPU manager of GPU-EvR. In Line 1-7, variables are initialized. After that, the GPU executes the applications in Line 8. In Line 9, if resource reallocation flag is set, the GPU manager starts GPU resource reallocation. In Line 10, the resource reallocation list and the reallocatable GPU resources are obtained. By using Equation 6, the GPU manager obtains the amount of resources to meet the timing requirement in Line 12. If the required amount of resources is less than or equal to the reallocatable resources, the GPU manager assigns required resources and updates reallocatable resources in Line 14-15. However, in Line 16-18, if reallocatable resources are smaller than the required amount, current reallocatable resources are assigned to current application in resource reallocation list. In Line 19-20, the GPU manager reallocates GPU resources and updates the resource reallocation list. This process will be repeated until all applications in the resource reallocation list are processed or there are no more resources for reallocation.

The functions $CreateReallocList()$ has a complexity of $O(|A'|)$. The while-loop in Line 11-20 has a complexity of $O(|ReallocList|)$ and $GetNewResource()$ has a complexity of $O(|P_{tot}|)$. Therefore, the overall complexity of the GPU manager algorithm is given by $O(|A'| + |ReallocList| * |P_{tot}|) = O(n^2)$.

Application Name	Dwarves	Domains
Leukocyte	Structured Grid	Medical Imaging
Heart Wall	Structured Grid	Medical Imaging
CFD Solver	Unstructured Grid	Fluid Dynamics
LU Decomposition	Dense Linear Algebra	Linear Algebra
HotSpot	Structured Grid	Physics Simulation
Back Propagation	Unstructured Grid	Pattern Recognition
Kmeans	Dense Linear Algebra	Data Mining
Breadth-First Search	Graph Traversal	Graph Algorithms
SRAD	Structured Grid	Image Processing
Streamcluster	Dense Linear Algebra	Data Mining
PathFinder	Dynamic Programming	Grid Traversal
Gaussian Elimination	Dense Linear Algebra	Linear Algebra
B+ Tree	Graph Traversal	Search

TABLE I: Rodinia benchmark Suite [3]

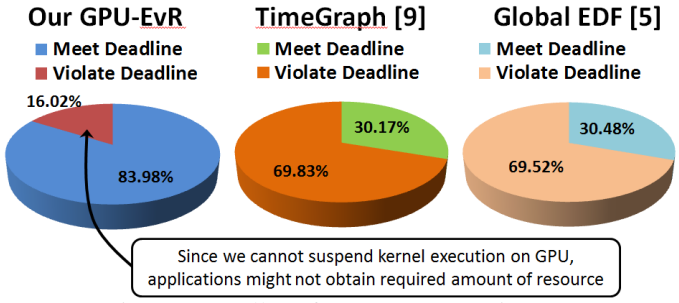

Fig. 3: Overall performance comparison

V. Experimental Results

We have extensively evaluated our framework by comparing it to several existing frameworks. In our experiments, we have used Nvidia's Tesla K20m graphic card that has a Kepler GK110 GPU (TSMCs 28nm manufacturing process).

In order to evaluate our framework with realistic workloads, we classify the application type based on application domain and dwarves [1]. Dwarves are common computation and communication pattern of the high performance parallel applications. Table I describes dwarves and application domains of Rodinia benchmark suite. We classify priority of the Rodinia benchmark Suite [3] based on application domains and dwarves. For high priority application, since most of image processing requires real-time behaviour, image processing benchmark applications are classified as high priority application. The benchmark applications which have simple behaviour (i.e. graph traversal, vector computation) are classified as low priority. Remaining benchmark applications are classified as medium priority. The classification results are as follows:

- High priority: Leukocyte, Heart Wall, HotSpot, SRAD;
- Medium priority: Back Propagation, PathFinder, Kmeans, Streamcluster;
- Low priority: Breadth-First Search, B+ Tree, Gaussian Elimination, LU Decomposition, CFD Solver;

A. Random injection of applications

GPU-EvR is compared to *TimeGraph* [9] and *Global-EDF* [5] scheduling frameworks. Figure 3 describes overall performance of the timing guarantee. During simulation, applications are randomly selected and injected into our experimental platform. Delays between applications are also randomly selected within 0.5 second window. We observe that more applications may meet timing requirement with our framework. The major reason for the improvement is the execution mechanism of the applications. Both *TimeGraph* and *Global EDF* scheduling frameworks handle one application at a time. However, our framework may handle multiple applications at the same time. Also, our framework can allocate more GPU resources to higher priority applications. Additionally, we see that there are applications which may not meet timing requirement with our framework. Since on-going applications may not be suspended, the scheduler may not allocate enough GPU resources to the higher priority applications.

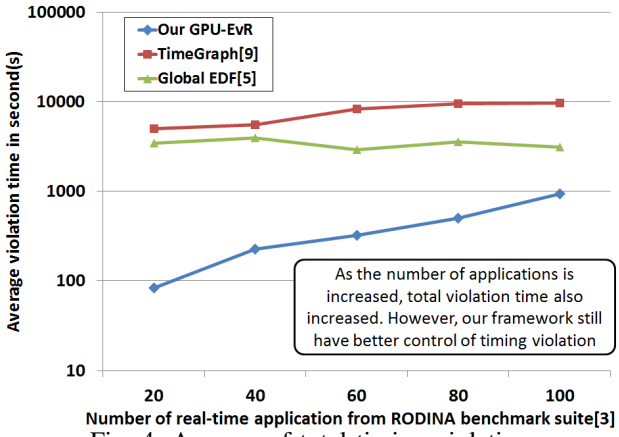

Fig. 4: Average of total timing violation

Figure 4 shows the total timing violation. In our framework, we observe that the average timing violation of the injected applications is increased as the number of applications is increased. However, total timing violation of our framework is much less than *TimeGraph* and *Global EDF*. It is observed from the figure that our framework has better control of timing. Since our framework allows concurrent execution of applications and allocates more GPU resources to higher priority applications, our framework is able to minimize timing violation.

Figure 5 shows the resource utilization and the average power per application. During the experiment, power is measured by using Nvidia's NVML library [15]. In Figure 5, lines represent the average power per application and bars represent the GPU resource utilization. As can be seen from the figure, our framework uses less GPU resources and power per application in 28nm technology. The reason for the efficiency is that our framework tries to find and allocate minimum amount of GPU resources to meet the timing requirement.

B. Scalability of the scheduling platform

We have also evaluated scalability of our GPU-EvR compared to [5], [9]. During our experiment, the number of injected applications in one period is varied from one to fifteen. Figure 6 shows the average number of applications which can meet the timing requirement. When the number of the injected applications is scaled from one to three, all the applications may meet timing requirement with our framework. After that, until the number of the injected applications is increased to seven, our framework can guarantee timing requirement for 2.5 applications. As the injected number of applications is greater than seven, applications are congested and performance of our framework is affected by the congested applications. However, from the figure,

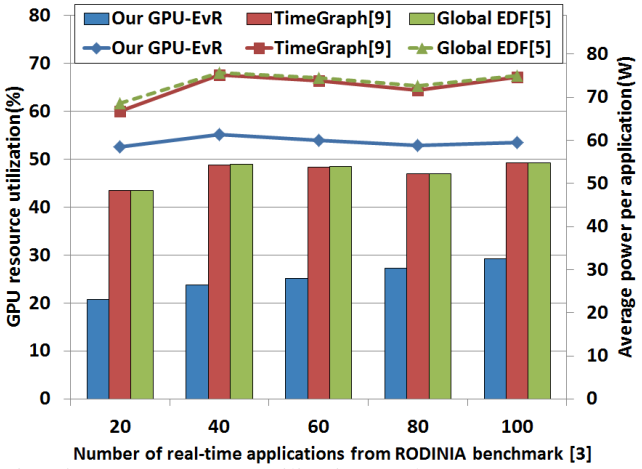

Fig. 5: GPU resource utilization and average power per application in 28nm technology

we observe that more applications are able to meet their timing requirement with our framework. This is because our framework provides more fine-grained control of GPU resources management by actively involving in mapping between thread blocks of the applications and the streaming multiprocessors.

VI. Conclusion

In this paper, we addressed a novel GPU scheduling framework for the event based real-time embedded systems (GPU-EvR). The presented scheduling framework consists of two levels which are the workload manager and the GPU manager. The workload manager selects the application from waiting queues based on application's priority and the GPU manager supports preemption by a run-time resource management algorithm. By using our execution and timing model, GPU-EvR may allocate more GPU resources to higher priority applications and estimate the amount of resources to meet the timing requirement. We have evaluated our framework by comparing the performance of our solution with *TimeGraph* and *Global EDF* scheduling frameworks. The results show that GPU-EvR is able to guarantee up to 6.4 times as many applications and a better control of timing violation. Applications use up to 20% less power under GPU-EvR. In addition, compared to other frameworks, the results clearly show that our framework may manage concurrent execution of multiple applications very efficiently.

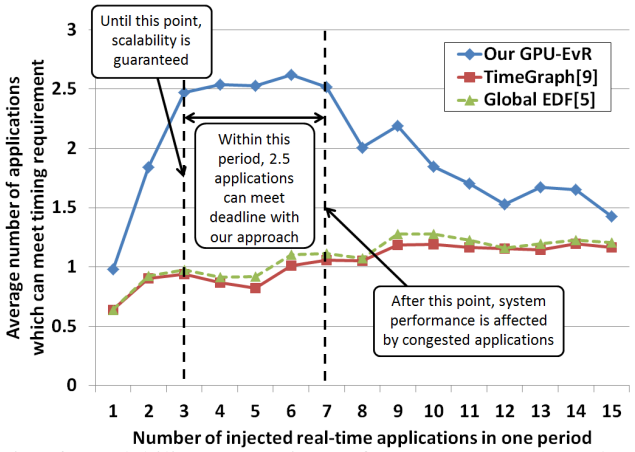

Fig. 6: Scalability comparison of GPU-EvR compared to [5], [9]

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. "The Landscape of Parallel Computing Research: A View from Berkeley". *EECS Department, University of California, Berkeley*, 2006.

[2] C. Basaran and K.-D. Kang. "Supporting Preemptive Task Executions and Memory Copies in GPGPUs". *Euromicro Conference on Real-Time Systems (ECRTS'12)*, 2012.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing". *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, pages 44–54, 2009.

[4] U. Devi and J. Anderson. "Tardiness bounds under global EDF scheduling on a multiprocessor". *Real-Time Systems Symposium (RTSS'2005)*, 2005.

[5] G. A. Elliott and J. H. Anderson. "Globally scheduled real-time multiprocessor systems with GPUs". *International Conference on Real-Time and Network Systems (RTNS'10)*, 2010.

[6] G. A. Elliott and J. H. Anderson. "An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems". *International Conference on Real-Time Networks and Systems (RTNS'11)*, pages 15–24, 2011.

[7] G. A. Elliott and J. H. Anderson. "Real-World Constraints of GPUs in Real-Time Systems". *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, 2011.

[8] S. Kato, J. Aumiller, and S. Brandt. "Zero-copy I/O processing for low-latency GPU computing". *International Conference on Cyber-Physical Systems (ICCPS'13)*, 2013.

[9] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. "TimeGraph: GPU scheduling for real-time multi-tasking environments". *USENIX Annual Technical Conference (USENIX ATC'11)*, 2011.

[10] Y. Liang, Z. Cui, S. Zhao, K. Rupnow, Y. Zhang, D. Jones, and D. Chen. Real-time implementation and performance optimization of 3d sound localization on gpus. *Design, Automation Test in Europe Conference Exhibition (DATE'12)*, pages 832–835, 2012.

[11] R. Membarth, J.-H. Lupp, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Dynamic task-scheduling and resource management for GPU accelerators in medical imaging". *International Conference on Architecture of Computing Systems (ARCS'12)*, pages 147–159, 2012.

[12] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, and Y. Deng. "Evaluating the potential of graphics processors for high performance embedded computing". *Design, Automation Test in Europe Conference Exhibition (DATE'11)*, pages 1–6, 2011.

[13] NVIDA. "CUDA C/C++ Streams and Concurrency". 2011.

[14] NVIDIA. "CUDA C Programming Guide", 2012.

[15] NVIDIA. "NVML API REFERENCE MANUAL", 2012.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. "A Survey of GeneralPurpose Computation on Graphics Hardware". *Computer Graphics Forum*, 2005.

[17] C. Thompson, S. Hahn, and M. Oskin. "Using modern graphics architectures for general-purpose computing: a framework and analysis". *International Symposium on Microarchitecture (MICRO-35)*, pages 306–317, 2002.

[18] B. C. Ward, G. A. Elliott, and J. H. Anderson. "Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols". *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, pages 280–289, 2012.

[19] Y. Zhu, Y. Deng, and Y. Chen. "Hermes: An integrated CPU/GPU microarchitecture for IP routing". *Design Automation Conference (DAC'11)*, pages 1044–1049, 2011.