JAWS: A JavaScript Framework for Adaptive CPU-GPU Work Sharing

Xianglan Piao[†] Channoh Kim[†] Younghwan Oh[†] Huiying Li[†]
Jincheon Kim[§] Hanjun Kim[‡] Jae W. Lee[†]

[†]Sungkyunkwan University, Suwon, Korea
{hlpark, channoh, garion9013, hyb2uty, jaewlee}@skku.edu

[§]Company100, Seoul, Korea
jck@company100.com

[‡]POSTECH, Pohang, Korea
hanjun@postech.ac.kr

Abstract

This paper introduces JAWS, a **J**avaScript framework for **a**daptive **w**ork **s**haring between CPU and GPU for data-parallel workloads. Unlike conventional heterogeneous parallel programming environments for JavaScript, which use only one compute device when executing a single kernel, JAWS accelerates kernel execution by exploiting both devices to realize full performance potential of heterogeneous multicores. JAWS employs an efficient work partitioning algorithm that finds an optimal work distribution between the two devices without requiring offline profiling. The JAWS runtime provides shared arrays for multiple parallel contexts, hence eliminating extra copy overhead for input and output data. Our preliminary evaluation with both CPU-friendly and GPU-friendly benchmarks demonstrates that JAWS provides good load balancing and efficient data communication between parallel contexts, to significantly outperform best single-device execution.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Classification—JavaScript

Keywords Web browser; JavaScript; data parallelism; GPU; work sharing; scheduler; multi-core; heterogeneity

1. Introduction

As more and more applications are deployed on the web, JavaScript has become a mainstream programming environment that enables heavyweight web applications. Today it is common to run complex, compute-intensive applications on the web browser such as media players, 3D renderer, and online games. With widespread adoption of heterogeneous processors, comprised of both CPUs and GPUs, JavaScript is called upon to embrace heterogeneity as well as parallelism in processing elements to execute a wide variety of parallel workloads efficiently. To this end, several parallel programming frameworks have been recently proposed to accelerate data-parallel workloads, including WebCL [1].

Although designed for heterogeneous parallel computing, these frameworks use only one compute device–either CPU or GPU, but not both–when executing a kernel, typically selected at kernel invocation. Since one device executes the kernel, the other device remains mostly idle, which leaves hardware resources underutilized. This resource underutilization problem is more pronounced on embedded platforms with limited hardware resources.

While recent proposals to exploit multiple devices concurrently to execute the kernel demonstrate promising results for conventional languages (e.g., C) [3, 5], JavaScript poses unique challenges in realizing such a framework. Web Workers [2] are the only thread-like programming construct that is universally supported by JavaScript engines, but with shared-nothing semantics. In addition, the data communication cost between parallel contexts (Workers) is extremely high in JavaScript. Low communication bandwidth with no shared memory support significantly increases the overhead of distributing input data to and merging the kernel output from multiple parallel contexts. High communication latency degrades the effectiveness of the work dispatching loop.

To address these challenges, we introduce JAWS, the *first* JavaScript framework for efficient CPU-GPU work sharing for data-parallel workloads. To achieve robust performance in a high-latency JavaScript environment, we devise an efficient online work partitioning algorithm without requiring offline training runs. To efficiently merge the output chunks from both devices, JAWS supports JavaScript-level shared arrays between Web Workers, hence eliminating extra copy overhead. Our preliminary evaluation demonstrates that JAWS can outperform best single-device execution for both CPU-friendly and GPU-friendly programs.

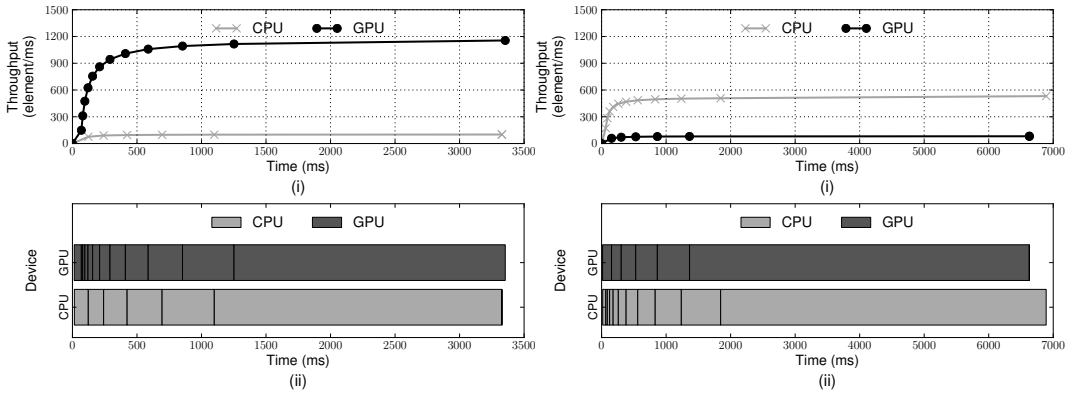
2. JAWS Runtime System

Execution Model. JAWS implements a task scheduler, which partitions the kernel input into chunks and distributes them to a multicore CPU (running a JavaScript kernel on multiple Web Workers) and a GPU (running a WebCL kernel) for concurrent execution. A chunk is formed by taking a contiguous subset of the flattened input data [5], specified by a pair of array indices pointing to the first and last elements of the subset, as we focus on array-based data parallel workloads. To effectively communicate input and output data between workers by *references*, instead of values, JAWS allocates shared arrays accessible by all workers (including one worker managing GPU execution). Thus, to dispatch a chunk of work, the scheduler needs to send only pairs of array indices instead of sending the entire data. The scheduler dynamically adapts the chunk size for each device to minimize the overhead of the work dispatching loop. Once execution of a chunk is finished on a compute device, the device writes the produced output chunk into the shared output buffer and signals the task scheduler to fetch a new task. This process continues until the task queue becomes empty.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

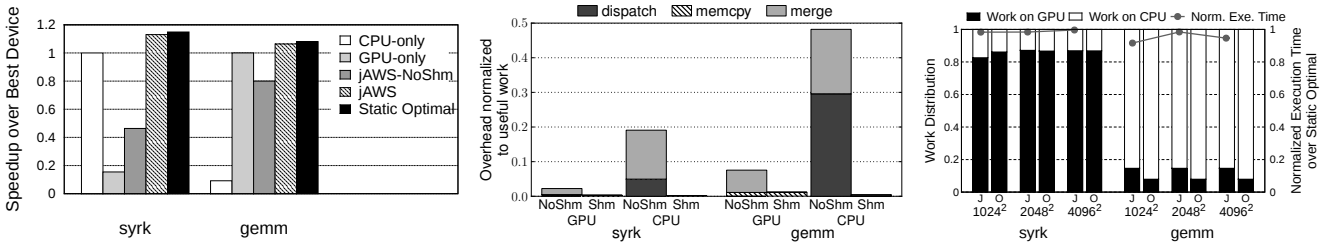
Copyright is held by the author/owner(s).

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688525



(a) gemm (GPU friendly) (b) syrk (CPU friendly)

Figure 1: Runtime behavior of the proposed algorithm for gemm and syrk



(a) Overall Speedup over Best Device (b) Breakdown of Execution Overhead (c) Work Distribution

Figure 2: Performance of JAWS running two Polybench benchmarks: gemm and syrk

Work Partitioning Algorithm. The task scheduler of JAWS sets the initial chunk size of both devices to be a predetermined fraction (denoted by α) of the work size and dispatches a chunk to each device. The scheduler *multiplicatively* increases the chunk size by a constant factor (denoted by β) until the throughput becomes stabilized. Note that the throughput of each device generally increases with an increasing chunk size and gets saturated beyond a certain threshold. Once the throughput of either device is saturated, the scheduler partitions the remaining work in proportional to each device's throughput and dispatches it to both devices. Figure 1 captures runtime behaviors of the algorithm with two benchmarks used for evaluation: gemm and syrk. The two stacked figures for each benchmark show (i) throughput, and (ii) execution time line for chunk execution, respectively. As time goes on, the chunk size increases multiplicatively by a factor of β until throughput gets saturated. Once this point is reached, the remaining work is partitioned and distributed to both devices. The algorithm demonstrates good load balancing, hence yielding good performance.

3. Preliminary Evaluation

JAWS is evaluated on two 6-core Intel Xeon E5645 CPUs clocked at 2.40GHz with 12GB RAM and a 96-core Nvidia Quadro 600 GPU clocked at 1.28GHz with 1GB of global memory. We use two Polybench benchmarks [4]: syrk (CPU-friendly) and gemm (GPU-friendly). The input size is set to (2048, 2048) for both.

Figure 2(a) shows the overall performance speedup over sequential execution. JAWS and JAWS-NoShm denote the results for JAWS with and without shared array support, respectively, and the shared array is turned on for the static optimal case. While JAWS-NoShm suffers high slowdown due to communication overhead, JAWS increases performance close to the static optimal algorithm for both CPU-friendly and GPU-friendly benchmarks.

Figure 2(b) breaks down the runtime overhead into three components. The shared array (labeled "Shm") eliminates most of the overhead from (task) dispatch and merge operations. The overhead of memory copy between the host (CPU) and the device (GPU) (denoted by memcpy) remains constant because the shared array does not eliminate this overhead. Finally, Figure 2(c) compares work distribution of JAWS (labeled "J") against that of the static optimal partitioning algorithm (labeled "O") with different input sizes. JAWS allocates chunks to both devices in proportional to their relative throughput, to yield good load balance, and hence near-optimal performance speedups.

Acknowledgments

This work was supported by the Ministry of Science, ICT & Future Planning (MSIP) under the "Global Excellent Technology Innovation R&D Program" (KEIT-10047038), "Research Project on High Performance and Scalable Manycore Operating System" (#14-824-09-011) and "IT Consilience Creative Program" (NIPA-2014-H0201-14-1001).

References

[1] WebCL Standard. URL http://www.khronos.org/webcl/.

[2] Web Worker. URL http://www.w3.org/TR/workers.

[3] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *CF*, 2013.

[4] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of Innovative Parallel Computing (InPar)*, 2012.

[5] P. Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *CGO*, 2014.