Efficient CPU-GPU Work Sharing for Data-Parallel JavaScript Workloads

Xianglan Piao† Channoh Kim† Younghwan Oh† Hanjun Kim‡ Jae W. Lee†

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

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

ABSTRACT

Modern web browsers are required to execute many complex, compute-intensive applications, mostly written in JavaScript. With widespread adoption of heterogeneous processors, recent JavaScript-based data-parallel programming models, such as River Trail and WebCL, support multiple types of processing elements including CPUs and GPUs. However, significant performance gains are still left on the table since the program kernel runs on only one compute device, typically selected at kernel invocation. This paper proposes a new framework for efficient work sharing between CPU and GPU for data-parallel JavaScript workloads. The work sharing scheduler partitions the input data into smaller chunks and dynamically dispatches them to both CPU and GPU for concurrent execution. For four data-parallel programs, our framework improves performance by up to 65% with a geometric mean speedup of 33% over GPU-only execution.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.2 [**Programming Languages**]: Language Classification—*JavaScript*

Keywords

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

1. INTRODUCTION

JavaScript is the default programming environment for browser-based web applications. As more and more applications are deployed on the web, the role of JavaScript has grown from a light-weight scripting language to a general-purpose programming framework that enables heavy-weight web applications. Besides, as heterogeneous processors comprised of both CPUs and GPUs are widely adopted, JavaScript is called upon to embrace heterogeneity as well as parallelism in processing elements to execute a wide variety of parallel workloads efficiently.

Copyright is held by the author/owner(s).
WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2577338.

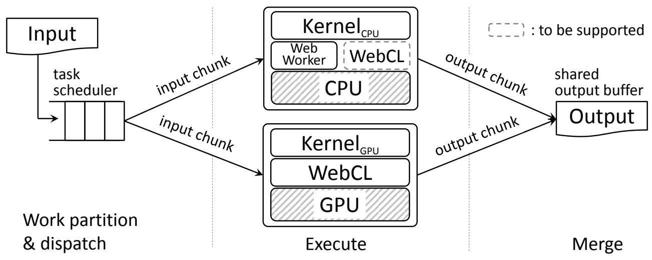

Figure 1: Work sharing execution model

Recently, several parallel programming frameworks have been proposed to accelerate data-parallel workloads, including River Trail [6] and WebCL [2]. Although both are designed for heterogeneous parallel computing, they use only one compute device—either CPU or GPU, but not both—when executing a kernel. This leaves hardware resources underutilized and potential performance gains on the table.

To address this limitation, we propose a new JavaScript framework for efficient work sharing between CPU and GPU for data-parallel workloads. The work sharing scheduler partitions the input data into small chunks to create *tasks*, and automatically dispatches them to both CPU and GPU. The CPU and the GPU execute tasks concurrently and produce output chunks. To efficiently merge the output chunks from both devices, the framework supports JavaScript-level shared memory between the two devices, hence eliminating extra copy operations. We prototype the framework on WebKit, a popular browser engine that powers many production-grade web browsers. The preliminary performance results look promising with a maximum speedup of 65% and a geometric mean speedup of 33% over GPU-only execution for four data-parallel JavaScript programs.

2. DESIGN AND IMPLEMENTATION

Figure 1 illustrates the execution model of the proposed work sharing framework. Work sharing uses a globally shared task queue to distribute tasks to multiple compute devices [7]. In our setup, a multi-core CPU and a GPU are two available compute devices. Also, a task is formed by taking a fixed-size subset of the input data, 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. Once a task is dispatched and executed on a compute device, the device writes the produced output chunk into the shared buffer and signals the task scheduler to fetch a new task. This process continues until the task queue becomes empty. Note that, no explicit output merging process is necessary since both devices write to the shared output buffer.

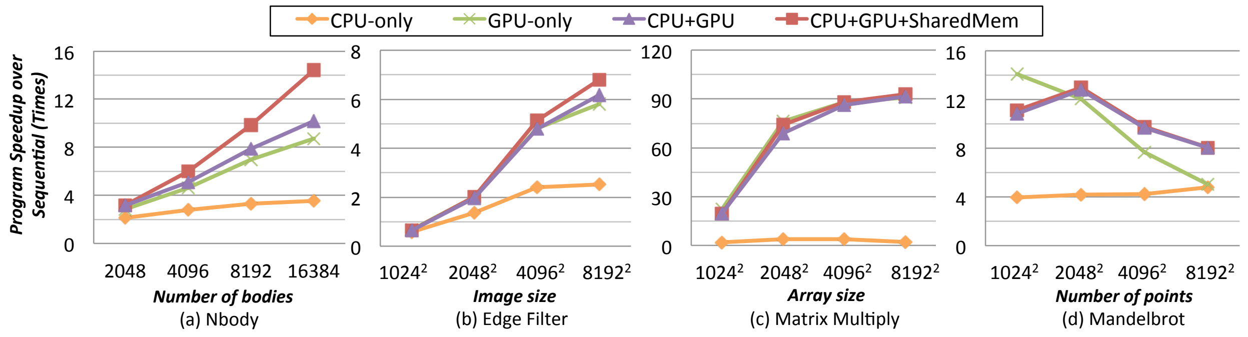

Figure 2: Performance speedups of four programs normalized to sequential execution varying input sizes.

The parallel execution framework builds on Web Worker [4], with which we create multiple parallel contexts in JavaScript. For a machine with N CPU cores, the framework spawns N+1 Web Workers: N for CPU execution and one for GPU execution. The N Workers for CPU execute the kernel written in JavaScript concurrently; The Worker for GPU invokes an OpenCL version of the same kernel through WebCL binding. For communication between the task scheduler on the main thread and the compute devices, we use JavaScript event handlers such as `postMessage` and `onmessage`.

For efficient data communication we modify the browser engine to support shared memory among parallel contexts. Note that, the Web Worker API is based on the shared-nothing model and requires explicit message passing for communication between the main thread and a worker thread. This shared memory support is particularly valuable in merging the produced output chunks from multiple compute devices; otherwise, it would be necessary to copy the output chunk back to the main thread whenever a task is finished.

The chunk size determines the granularity of a task and is an important performance parameter. A smaller chunk generally leads to better load balancing among compute devices at the cost of higher dispatching overhead. In a non-JavaScript setup non-uniform chunk sizes for different compute devices are reported to be beneficial [7]. We assume a uniform chunk size for both CPU and GPU and leave the exploration of non-uniform chunk sizes for future work.

3. EVALUATION

We evaluate the proposed framework on Intel i5-3330 CPU with 4 cores clocked at 3.0GHz with 4GB RAM and Nvidia GeForce GT 620 GPU clocked at 1.62GHz with 1GB of global memory. Table 1 summarizes the four data-parallel JavaScript programs used for evaluation. The same fixed chunk size is used for both CPU and GPU.

Program	Chunk size	Source
Nbody	256 Bytes	WebKit-WebCL [5]
Edge Filter	64×64 Bytes	River Trail [3]
Matrix Multiply	256×256 Bytes	Nvidia OpenCL SDK [1]
Mandelbrot	256×256 Bytes	River Trail [3]

Table 1: Benchmark programs

Figure 2 shows the program speedups normalized to sequential JavaScript execution with varying input sizes. CPU-only and GPU-only refer to the speedup numbers with only using either CPU or GPU, respectively; CPU+GPU refers to CPU-GPU work sharing execution *without* shared memory, and CPU+GPU+SharedMem *with* shared memory. With a large input (i.e., enough work to do), CPU+GPU+SharedMem eventually outperforms the other three alternatives except for `Matrix Multiply` for which it performs slightly worse than GPU-only. In general, the work sharing scheme works well for those programs whose performance gap between CPU-only and GPU-only is relatively small; otherwise, GPU-only execution without the overhead of managing CPU-GPU cooperative execution would likely perform better.

For both `Nbody` and `Edge Filter` the shared memory support results in significant performance boost. The performance benefit is more pronounced in `Nbody` since it is necessary to propagate (copy) the position and velocity arrays to all compute devices every iteration. `Matrix Multiply` is a very GPU-friendly program, and its performance gain by offloading computation to CPU is relatively small. Finally, for `Mandelbrot` CPU+GPU+SharedMem performs best for large inputs. However, the difference between CPU+GPU and CPU+GPU+SharedMem is rather small due to a low volume of communication. Also, the speedups decrease as input size increases since the fraction of the sequential portion of the program (drawing) in execution time increases.

4. CONCLUSION

In this paper we propose a novel JavaScript parallel execution framework that enables efficient CPU-GPU work sharing. Our prototype of the framework built on WebKit demonstrates promising results. In the future, we plan to extend the framework to support OpenCL kernels on CPU and non-uniform chunk sizes for different compute devices.

5. ACKNOWLEDGMENTS

This work was supported by the Korean Evaluation Institute of Industrial Technology funded by the Ministry of Science, ICT & Future Planning (KEIT-10047038). Hanjun Kim was partly supported by the MSIP Korea under the IT Consilience Creative Program (NIPA-2013-H0203-13-1001).

6. REFERENCES

[1] Nvidia Developer Zone. `https://developer.nvidia.com/opencl`.
[2] Parallel Computing on the Web. `http://webcl.nokiaresearch.com`.
[3] River Trail. `http://github.com/RiverTrail/RiverTrail`.
[4] Web Worker. `http://www.w3.org/TR/workers`.
[5] WebCL for WebKit. `http://github.com/SRA-SiliconValley/webkit-webcl`.
[6] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in JavaScript. In *OOPSLA*, 2013.
[7] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ICS*, 2010.