Runtime Support for Concurrent Execution of Overdecomposed Heterogeneous Tasks

Jaemin Choi
University of Illinois Urbana-Champaign
Urbana, Illinois
jaemin@acm.org

Laxmikant V. Kale (advisor)
University of Illinois Urbana-Champaign
Urbana, Illinois
kale@illinois.edu

ABSTRACT

With the rise of heterogeneous systems in high performance computing, how we utilize accelerators has become a critical factor in achieving the optimal performance. We explore several issues with using accelerators in Charm++, a parallel programming model that employs overdecomposition. We propose a runtime support scheme that enables concurrent execution of heterogeneous tasks and evaluate its performance. Using a synthetic benchmark that utilizes busy-waiting to simulate workload, we observe that the effectiveness of the runtime support varies with the application characteristics, with a maximum speedup of 4.79x. With a two-dimensional five-point stencil benchmark designed to represent a realistic workload, we obtain up to 2.75x speedup.

1 INTRODUCTION

1.1 The Charm++ Programming Model

Charm++ is a migratable objects and task based parallel programming model with an adaptive runtime system. The problem domain is decomposed into objects containing data and methods that manipulate the data. The objects communicate with each other by invoking methods in a message-driven fashion. The execution of an object's method corresponds to a task: the unit of execution that is scheduled by the runtime on a processing element (PE, typically a CPU core). Charm++ adopts *overdecomposition*, where there are generally many more objects than the number of PEs. This empowers the runtime system to make intelligent scheduling decisions and achieve overlap of computation and communication, as well as perform load balancing [1].

1.2 Overdecomposition and Heterogeneous Tasks

A common method used to leverage heterogeneity is to offload computation to the device, where we assume CUDA as the medium of execution. This is done in Charm++ by placing the API calls in a method. We name the execution of such a method a *GPU task* and a method execution on the CPU a *CPU task*. An object could have multiple such tasks.

With overdecomposition, it is highly likely that multiple GPU tasks will execute at the same time. This probability increases with the number of PEs as more objects are executed in parallel, as well as the duration of the kernels. A recent feature in CUDA, *concurrent kernel execution*, allows kernels in separate streams to execute concurrently. The Charm++ programmer should take advantage of this feature and assign each object one or more streams to enable multiple GPU tasks to run simultaneously on the device.

Waiting for the completion of a GPU task via `cudaStreamSynchronize()` is a widely used method. However, this call blocks the CPU and does not allow any other task (including other GPU tasks) to be executed until the GPU work is done. As a result, the degree of concurrency on the device becomes limited; the maximum number of GPU tasks that could execute concurrently is equal to the number of PEs.

As such, concurrent execution of heterogeneous tasks is a critical factor in performance. Our work focuses on enabling it through the support of the runtime system.

SC'17, November 2017, Denver, Colorado USA
2017. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

2 RUNTIME SUPPORT

We propose a runtime support scheme that is designed to address the heterogeneous execution problem by integrating recent features of CUDA. The candidate features are callbacks and events, both of which require modification of the Charm++ runtime. We choose to utilize CUDA events, because the single thread generated by CUDA runtime to handle callbacks becomes a bottleneck as we scale the number of PEs.

CUDA events allow the user to poll for the completion of a specific operation in a stream, rather than waiting for it as with `cudaStreamSynchronize()`. Nevertheless, it is impractical for the user to create a separate polling method, as he/she would have to integrate it somewhere in the flow of execution and determine the polling frequency, both of which are unclear.

We modify the Charm++ runtime so that the scheduler polls for GPU event completion before executing any task. Each PE has a queue that the scheduler polls, where an event is pushed into when a GPU operation of interest is posted. When the scheduler sees an event marked as complete, it invokes the associated Charm++ callback.

A potential drawback of this approach is the overhead caused by polling; checking the event queue could delay the execution of a pending task. We minimize this delay by stopping the queue traversal as soon as we find an event that is not complete, as this implies that none of the following events are complete.

3 EXPERIMENTAL RESULTS

We evaluate the performance of our proposed scheme using two benchmarks, `busywait` and `stencil2d`. They were tested on a single compute node of the Titan supercomputing system located at the Oak Ridge Leadership Computing Facility. Each compute node consists of a 16-core AMD Opteron 6274 CPU, 32GB DDR3 memory, and a Nvidia Tesla K20X GPU. We utilize 8 out of the 16 available CPU cores as two consecutive cores share a floating point scheduler.

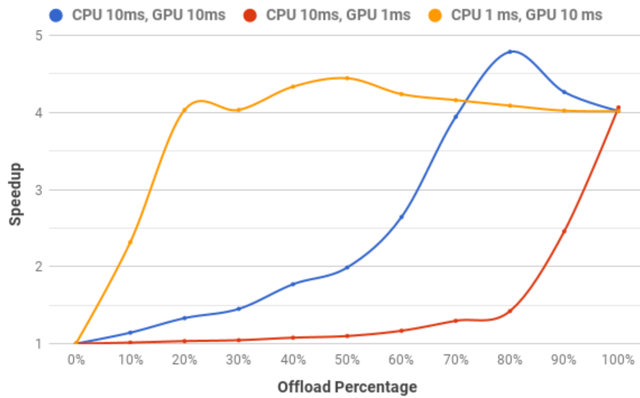

Figure 1: Speedup of `busywait` Benchmark

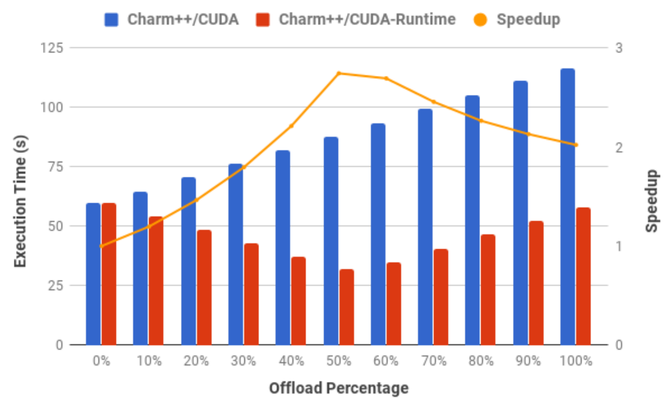

Figure 2: Performance of `stencil2d` Benchmark

3.1 `busywait`

Each object in the `busywait` benchmark executes one core task that simulates work via busy-waiting a fixed amount of time. We vary the percentage of objects that busy-wait on the GPU (*offload percentage*), and the durations of busy-waiting for each type of tasks. This benchmark was designed to validate the proposed scheme and identify how its effectiveness would vary in different applications.

The total number of objects is set to 128, resulting in 16 objects per PE. We vary the offload percentage from 0% to 100% in steps of 10%. Each object performs 100 iterations, and the number of threads utilized per GPU task is set to 512. This allows 32 kernels to execute concurrently, which is the maximum allowed by the device.

Figure 1 shows the speedup achieved with the proposed runtime support, where we see up to 4.79 times speedup compared to the use of CUDA without it (i.e. using `cudaStreamSynchronize()`). We see a broader range of offload percentages that exhibit good speedups when the CPU tasks execute much faster than the GPU tasks (in yellow) and a narrower range when this relation is reversed (in red). This is because being able to overlap more GPU tasks than the number of PEs becomes more effective with longer GPU tasks.

The result suggests that the effectiveness of the proposed scheme depends on the characteristics of the application, such as the percentage of computation offloaded to the device and the durations of the tasks.

3.2 `stencil2d`

We adopt the `stencil2d` benchmark to evaluate the performance of the proposed runtime support under a more realistic workload. The two-dimensional simulation space is decomposed into blocks, each of which belong to an object. The objects first communicate the halo regions with their neighbors, and then perform a five-point stencil operation either on the CPU or the GPU. This is repeated a given number of iterations.

The global simulation space consists of 16384 x 16384 double-precision floating point values with blocks of size 512 x 512, resulting in a total object count of 1024. Each stencil CUDA kernel uses 64 threads, as each thread computes a 64 x 64 sub-block. This enables 32 kernels to execute concurrently on the device. Also note that the stencil kernel for the GPU is optimized to transfer only the halo regions to/from the device at each iteration.

Figure 2 compares the performance of the benchmark with and without support from the Charm++ runtime, colored in red and blue, respectively. The stencil operations on the GPU are about 2 times slower than on the CPU, which could be inferred from the offload percentages of 0% and 100%. With the runtime support, however, we are able to achieve overlap of heterogeneous tasks as well as full concurrency of GPU tasks, allowing the benchmark to perform much better. A maximum speedup of 2.75 is achieved when the offload percentage is 50%.

4 CONCLUSION

We have explored how heterogeneity affects task execution in Charm++, a parallel programming model that utilizes overdecomposition. While a limited degree of concurrent execution of GPU tasks can be achieved with the use of streams and concurrent kernel execution, true concurrent execution of heterogeneous tasks require support from the runtime system. We have seen from the benchmark results that our proposed scheme is indeed able to realize good speedups.

ACKNOWLEDGMENTS

The author would like to thank Michael P. Robson and Ronak Buch at University of Illinois Urbana-Champaign for providing valuable inputs and feedback.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

The author is grateful to Cisco Systems Inc. for funding support (gift award CG 587589) and for access to its Arcetri cluster.

REFERENCES

[1] J. Lifflander, G. C. Evans, A. Arya, and L. V. Kale. 2012. Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2404–2413. https://doi.org/10.1109/IPDPSW.2012.297