Runtime Coordinated Heterogeneous Tasks in Charm++

Michael P. Robson, Ronak Buch, Laxmikant V. Kale
University of Illinois at Urbana-Champaign
{mprobson, rabuch2, kale}@illinois.edu

***Abstract*— Effective utilization of the increasingly heterogeneous hardware in modern supercomputers is a significant challenge. Many applications have seen performance gains by using GPUs, but many implementations leave CPUs sitting idle.**

In this paper, we describe a runtime managed system for coordinating heterogeneous execution. This system manages data transfers to and from GPU devices and schedules work across the computational resources of the system. The programmer need only tag methods and parameters to enable heterogeneous execution.

Using this system, we observe improvements in programmer productivity and application performance. For selected benchmarks, when using heterogeneous execution we observe speedups of up to 3.09x relative to using only the host cores or only the device.

***Index Terms*—Accelerator architectures, Parallel programming, High performance computing, Runtime**

I. Introduction

Many current supercomputers derive a majority of their compute power from accelerator devices. Nvidia GPUs and Intel Xeon Phis have already seen widespread adoption in many Top 500 machines. As the march to exascale continues, several new machines will derive a sizable portion of their overall FLOPS from GPUs. These include both the Summit system at ORNL and the Sierra system at LLNL. However, programming models and systems have been slow to adapt to this changing environment. In this paper, we examine an extension to the CHARM++ parallel programming library that enables coordinated execution of heterogeneous tasks. We focus on compute kernels developed for Nvidia GPUs using CUDA. Our framework automatically generates tasks from user-annotated functions that can be executed on either the host or device. This strategy ensures full utilization of available hardware and reduces computation time. In this paper we examine the heterogeneous performance of two mini applications, `stencil2d` and `md`.

II. Background and Related Work

CHARM++ [6] is a task based, asynchronous parallel programming framework with an adaptive runtime system (RTS). In CHARM++ programs, data is decomposed into logical units (chares) which are then mapped to hardware resources (PEs). Chares communicate and exchange data via messages that invoke asynchronous methods. The parallel structure and methods of CHARM++ programs are described in a charm interface file, which is parsed by the charm translator `charmxi` to generate code for the runtime. In this paper, we modify `charmxi` to generate both host and CUDA versions of the entry methods tagged for execution on different devices. It can be extended to generate code for any hardware platform, but these two targets are sufficient for our tests. We also augment the CHARM++ runtime, adding the capability to schedule heterogeneous work across the host and device based on a provided heuristic.

Graphical processing units (GPUs) are becoming prevalent in the HPC community, as is evident from their number over time in the Top 500. Originally intended as special purpose accelerators for graphics applications, they are now user programmable and often referred to as the "device" (as opposed to the CPU or "host" cores) due to their supplementary use in a system. A variety of languages and tools for GPU programming exist ([10], [7], etc.), but GPUs remain more difficult to program for than traditional host cores. Unlike CPUs, GPUs are made up of hundreds of lightweight cores grouped together into streaming multiprocessors (SMs). These SMs share critical resources, such as registers and shared memory. Collections of threads, called warps, are launched on these SMs and execute in lockstep. This unique design can lead to strong performance for some highly parallel applications, e.g. graphics, but can be hampered by its strict SIMD nature (for instance when encountering branch divergence in code). Data movement is also a concern since the GPU cannot directly access host memory. Therefore, data must be copied to the device before being used, which often limits performance due to the latency and bandwidth constraints associated with transferring data across the PCIe bus.

A similar approach to using runtimes in heterogeneous environments can be found in the StarPU programming library[1]. They also schedule tasks, called codelets, and automate data transfer dynamically across different hardware targets. However, StarPU does not have a mechanism to automatically generate kernels for different platforms as our work does. We distinguish ourselves from other task based run times such as OmpSs[5] by offering more generality, not requiring entire programs to be explicitly constructed as a DAG. Similar work has also been carried out in the context of OpenCL[9], [3] with great success, but we can extend our work to multiple nodes.

The authors of [4] propose a solution that divides work into fine grain tasks and enqueues them in a single location. This potentially allows for heterogeneous execution and dynamic

978-1-5090-3858-9/16/$31.00 © 2016 IEEE ESPM2 2016; Salt Lake City, Utah, USA; November 2016

load balancing. However, they use a work stealing approach with a persistent device kernel, instead of a central manager, and they do not show results for mixed CPU-GPU execution as presented in this paper. The Legion programming model [2] can also execute in heterogeneous environments using similar techniques to our approach.

III. Methodology

Our execution model builds upon the earlier work of GPU Manager [11], which handles the delegation and execution of CUDA kernels in the context of the asynchronous message-driven runtime of Charm++ . This allows us to focus our work on higher-level concerns, such as code generation and dynamic target selection in our framework.

A. Charm++ GPU Manager

The GPU Manager operates by registering GPU kernels to be managed with the runtime system. By having the runtime asynchronously invoke kernels when data is available on the device, we automate the overlap of data movement and execution as seen in Figure 1. Due to inherent asynchrony of Charm++ , it is important to ensure that blocking operations, such as `cudaHostMalloc`, are handled by the system and do not block in user code. GPU Manager also automates some tedious CUDA-related tasks, namely copying data to and from the device before and after kernel execution.

When using GPU Manager directly, the user must write an explicit CUDA kernel and denote buffers which need to be moved to and from the device. The programmer must also register a callback with the runtime, which is called when the kernel is finished and data has been copied back to the host. This step is necessary since the call to GPU Manager returns once the runtime has copied the CUDA buffers; it does not block until the kernel has finished. GPU manager coordinates data movement and kernel invocations through a FIFO queue. When a PE goes idle and enough time has passed, the runtime invokes a progress function to issue new requests to the GPU. At this time, GPU Manager attempts to offload data for a new kernel, launch a kernel with complete data on the device, and move data for the completed kernel back to the host. Finally, when the data for the completed kernel is fully copied back, GPU Manager invokes the user supplied callback to continue execution.

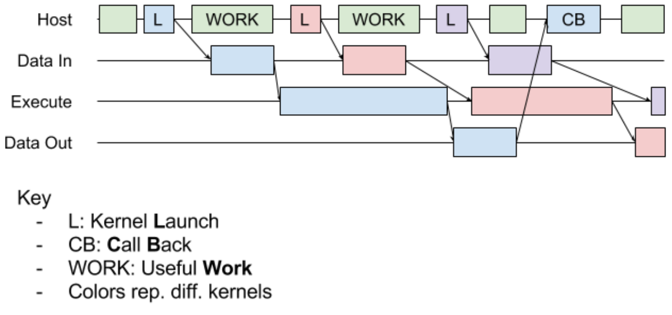

Fig. 1: GPU Manager

B. Accel Framework

The Accel Framework[8], or ACCEL, extends GPU Manager by automatically generating CUDA kernels from host code and dynamically deciding where entry methods should be executed.

ACCEL alleviates many of the programmer productivity problems associated with using GPUs effectively in parallel applications by virtue of its automatic kernel generation. This generation occurs only for entry methods annotated with the `accel` keyword. To improve performance, additional tags can be applied to methods, such as `splittable`, which allows methods to be split into several independent tasks, which can more fully utilize the many processors on a GPU. Inside `splittable` methods, `splitIndex` and `numSplits` variables are defined, analogous to the `threadIdx` and `blockDim` variables in CUDA. This differs from CUDA in that the code can be targeted to a variety of platforms. A full listing of other annotations can be found in [8].

ACCEL has a variety of strategies to determine where to execute particular entry methods. The strategy is passed to as a runtime argument. Example strategies include `+accelHostOnly`, `+accelDeviceOnly`, `+accelPercentDevice`, which specify a static division of work between the computing resources. In this paper, we manually sweep through different static divisions to observe the performance behavior of the various configurations. However, there are several available automated methods to find the best split, such as greedy strategies and hill climbing. Further description is outside the scope of this paper and is detailed in [8].

In order to maximize GPU utilization and avoid serialization, ACCEL tries to batch multiple device method calls into a single kernel launch. This batching occurs when a specified count is reached or a certain amount of time elapses. The `triggered` keyword informs the runtime system that the accelerated entry method (AEM) will be invoked on every chare and that all chares will invoke said entry method before any chare invokes it a second time. Programmers can also specify the number of threads to be used per block in a kernel launch instead of having the runtime automatically determine one.

It is beneficial for the RTS to minimize data movement and overlap it with computation when possible. Data movement is automatically overlapped with computation as described in Section III-A. Method parameters are automatically copied to the device, but are not copied back since the Charm++ model dictates that entry method parameters have no lifetime beyond the entry method. However, object data used in an `accel` annotated method must be marked as `readonly`, `writeonly`, or `readwrite` to indicate whether it should be copied in, out, or both. Additional annotations such as `shared` and `persistent` allow the user to control the lifetime of the data on the device. With these annotations, `charmxi` automatically generates code to move data to and from the device. The implObj variable seen in the code is required due to the lack of a proper Charm++ compiler since we require a handle to the chare object and its data.

The last token in Listing 1 specifies a callback invoked when the accel entry method is finished executing. It is used in the same way as in GPU Manager but is listed here instead of as an input or member variable due to parsing constraints. The callback is used to send messages to invoke other methods since CHARM++ messages cannot be sent from accelerator devices.

```
entry [triggered splittable(NUM_ROWS) accel]
    void doCalculation() [
  readonly:float matrix[DATA_BUFFER_SIZE]
      <implobj->matrix>,
  writeonly:float matrixTmp[DATA_BUFFER_SIZE]
      <implobj->matrixTmp>
] { ... } doCalculation_post;
```

Listing 1: Accelerated Entry Method Annotations

IV. RESULTS

We analyze performance for varying distributions of work between the host and device for two different applications, stencil2d, which implements a two dimensional stencil, and the more complex md, which simulates electrostatic molecular dynamics. In both applications, the main compute methods have been annotated with accel and other tuning parameters. Our tests vary the percentage of work allocated to the device from 0% to 100% in increments of 5%. Theoretically, hybrid computation will improve performance, since more hardware can be used, but data transfer and batching costs create performance impediments.

The experimental results were gathered on the Stampede supercomputer. In particular, we used the visualization nodes of the system, which each feature an NVIDIA K20 GPU and two Intel Xeon E5-2680 processors. All runs were performed on a single node of the system with 16 CHARM++ processing elements, matching the 16 cores in the node. We measured elapsed time from the start of the calculation to the end of the last error calculation for both applications. This does not include startup or other fixed costs.

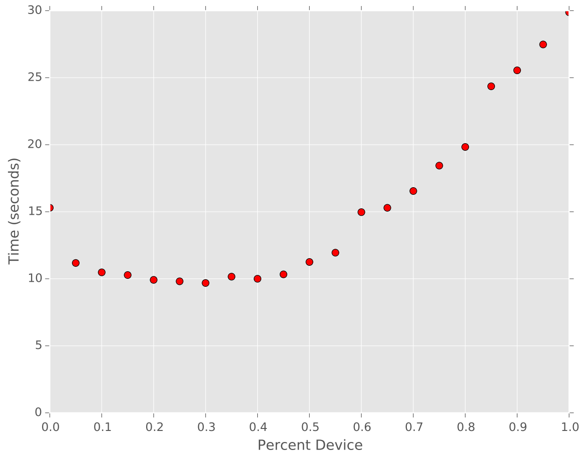

Fig. 2: Timing for stencil2d

A. *Stencil 2D*

stencil2d performs a single-precision weighted five point stencil. Given results use a 6144x6096 2D array decomposed into 24 tiles per dimension, a 254x254 section per element. For work performed on the GPU, the algorithm performs approximately 1.25 single-precision FLOP per transferred byte (10 FLOP/(1 float in + 1 float out)). As shown in Figure 2, this low FLOP/byte ratio causes the host only case to beat the device only case. Optimal performance occurs in the 30% device case.

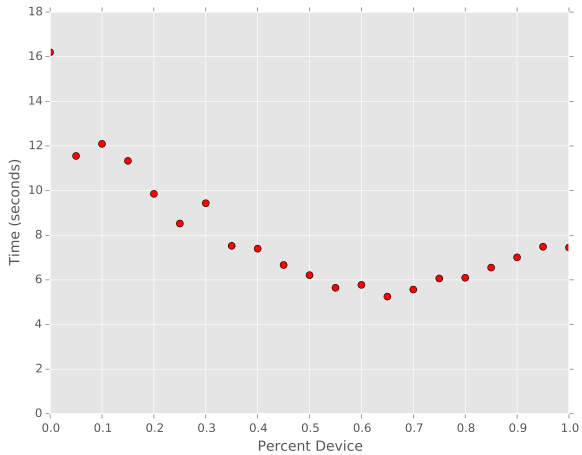

Fig. 3: Timing for md

B. *Molecular Dynamics*

md executes much faster in the device only case than in the host only case. Given results use a 5x5x5 3D array with 256 molecules per array element, a total of 32k molecules. The FLOP per byte ratio for md is higher than that of stencil2d since each particle has a relatively complex interaction with every other particle in the simulation, requiring distance, electrostatic force, position, and acceleration calculation and normalization. Optimal performance occurs in the 65% device case.

C. *Analysis*

For both selected applications, we observe an increase in performance when using both the host and the device as compared to using only the host or only the device.

There are some clear discontinuities in the performance of the chosen applications; see the jumps at 60% and 85% device in Figure 2. These are likely a performance artifact of the batching used in the Accel Framework. Since the GPU is a throughput-oriented device, launching an additional batch takes much longer than adding some work to an existing batch. This behavior is not seen for smaller allocations of work to the device because the host was spending more time on the work than the device, so it was the dominant term.

The timing data follows a “bathtub plot”, so termed because it is low in the middle and high on both sides. When performance follows this pattern, the goal is to set the parameters such that execution happens in the “floor” region.

As shown in Table I, the best configurations achieve speedups of between 1.46x and 3.09x relative to host only and device only configurations.

	Best Split	Host Only	Device Only
`stencil2d`	30% device	1.58x	3.09x
`md`	65% device	3.02x	1.46x

TABLE I: Speedup of Best Configuration Relative to Host/Device Only

D. Caveats

All applications do not benefit from a heterogeneous execution system. Even applications that are amenable to heterogeneous execution may not see benefit in all configurations. The most significant reason for this is data movement. Just as HPC applications can slow down when run on two nodes versus one node due to the effects of adding network communication, using a GPU can degrade performance unless the application amortizes the costs of data movement. Additionally, not all algorithms are well suited to run on the GPU. In particular, programs that make heavy use of branching, that cannot expose enough parallelism to fully utilize the GPU, or that are composed of a variety of disparate tasks do not perform well on GPU hardware.

However, large HPC applications often feature a variety of different kinds of work, so it is likely that some portion will improve when executed on heterogeneously.

V. Future Work

This work is a small survey of the initial implementation of CHARM++ support for heterogeneous compute environments. Our current solution to generating CUDA kernels, copying the entry method body directly into the kernel body with a few extensions, can be vastly improved. Using `splittable` allows performance to be greatly improved, but does not allow the user to make platform specific optimizations. One potential extension is to allow the user to explicitly provide optimized kernels for different platforms, as other runtimes, such as OmpSs, allow. We could also extend GPU manager to observe utilization and launch multiple kernels.

ACCEL currently provides mechanisms for the programmer to control data movement to and from the device (or in the case of persistence, residence). However, there is more work to be done here. For instance, adding support for GPU Direct, which would allow GPUs to directly communicate with each other, and reducing the number of copies of data made inside CHARM++ when transferring to and from the device. We also plan on taking into account data location and movement cost when making scheduling decisions to further minimize data movement. We anticipate that NVLink will partially alleviate some of these problems.

Finally, we plan on applying this work to automatically load balance heterogeneous applications at large scale. By extending the CHARM++ load balancing framework to support heterogeneous load balancing, we will balance work both across nodes and across the hardware resources in a node. This would enable CHARM++ programs to adapt to arbitrary hardware platforms of arbitrary size with minimal code changes.

VI. Conclusions

The use of accelerators in HPC has grown in recent years and will likely continue for the foreseeable future. While many HPC applications make use of accelerators to improve their performance, it remains difficult to fully utilize all hardware resources available on a machine.

In this paper, we describe a runtime for managing execution in heterogeneous environments. In contrast to common ways of using accelerators, in this scheme, the system handles data allocation, transfers, scheduling, and coordination across the heterogeneous hardware. This system requires minimal additional work from developers and is not tied to any specific accelerator platform.

We demonstrate the efficacy of this system for GPU execution using two CHARM++ applications. We achieve up to a 3.09x speedup relative to running the main computation solely on the host or device.

VII. Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of GPUs used for this research.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[3] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[4] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[6] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[7] D. Kirk et al. Nvidia cuda software and gpu parallel computing architecture.

[8] D. Kunzman. *Runtime support for object-based message-driven parallel applications on heterogeneous clusters*. PhD thesis, Dept. of Computer Science, University of Illinois, 2012. http://charm.cs.uiuc.edu/media/12-45/.

[9] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[10] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[11] L. Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.