

Towards Efficient Many-Task Computing on Accelerators in High-End Computing Systems

Scott J. Krieder*, Benjamin Grimmer*, Dustin Shahidehpour*, Jeffrey Johnson*
Justin M. Wozniak†, Michael Wilde†‡, Ioan Raicu*†

*Department of Computer Science, Illinois Institute of Technology

†MCS Division, Argonne National Laboratory

‡Computation Institute, University of Chicago

Abstract—Current software and hardware limitations prevent Many-Task Computing (MTC) workloads from leveraging hardware accelerators boasting Many Core Computing architectures. This work aims to address the programmability gap between MTC and accelerators, through the innovative CUDA middleware GeMTC. By working at the warp level, GeMTC enables heterogeneous task scheduling and 10x number of workers compared to CUDA. In order to span multiple accelerators across nodes, we have adopted the Swift parallel programming system, which can both support fine grained millisecond tasks and extreme scale supercomputers at 100K-cores.

Keywords-Many-Task Computing, Swift, GPGPU, CUDA

I. INTRODUCTION

This work aims to provide an integration between data-flow driven parallel programming systems (e.g. Many-Task Computing - MTC) and hardware accelerators [1] (e.g. NVIDIA GPUs, AMD GPUs, and the Intel MIC). MTC aims to bridge the gap between two computing paradigms, high throughput computing (HTC) and high-performance computing (HPC). MTC emphasizes using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds.[2] Swift is a particular implementation of the MTC paradigm, and is a parallel programming system that has been successfully used in many large-scale computing applications. [3] The scientific community has adopted Swift as a great way to increase productivity in running complex applications via a dataflow driven programming model, which intrinsically allows implicit parallelism to be harnessed based on data access patterns and dependencies. Swift is a parallel programming system that fits the MTC model, and has been shown to run well on tens of thousands of nodes with task graphs in the range of hundreds of thousands of tasks. This work aims to enable Swift to efficiently use accelerators (such as NVIDIA GPUs and Intel MIC) to further accelerate a wide range of applications, on a growing portion of high-end systems.

II. GEMTC

Currently CUDA developers may only have a maximum of 16 kernels running concurrently, one kernel per streaming multiprocessor (SM). The problem is that all kernels have to start and end at the same time, causing extreme inefficiencies

in heterogeneous workloads. By working at the warp level we trade local memory for concurrency and we are able to run up to 84 concurrent kernels. [4] This middleware allow independent kernels (MIMD style) to be launched and managed on many-core architectures that traditionally only support SIMD. [5] In Figure 1 Swift/T is calling the GeMTC API and passing tasks into memory on the device. Warp workers pick up those tasks, execute with the given parameters and place results on an outgoing result queue. This execution model is shown in further detail in Figure 2. Finally, Swift/T will poll the device return results from the result queue back to the appropriate task in the swift script. Figure 3 is a GK110 block diagram as presented by NVIDIA. [6] This diagram demonstrates how the current generation GPUs have O(10) Streaming Multiprocessors (SMX), O(100) Warps, and O(1000) cores.

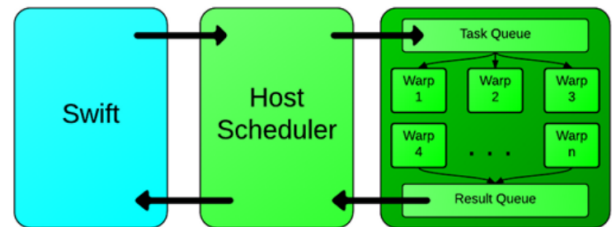


Fig. 1. Flow of a task through Swift/T and GeMTC.

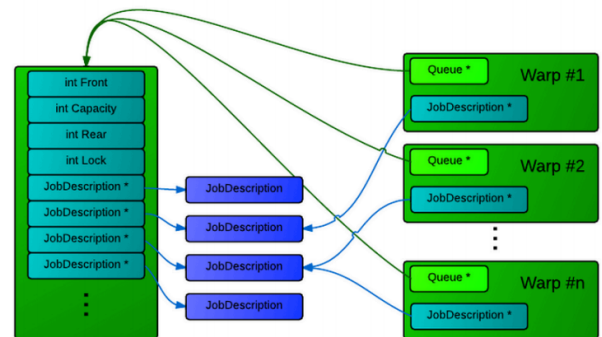


Fig. 2. Worker interaction with incoming work queue.



Fig. 3. Kepler GK110 full chip block diagram.

III. SWIFT/T

Swift is an implicitly parallel scripting language. By operating on data-flow driven scheduling of parallel tasks Swift is capable of extracting parallelism out of an application. The distributed executor eliminates centralized bottlenecks and an optimized compiler detects errors ahead of runtime for improved efficiency. Swift has been shown to scale over 100k cores and is portable to most MPI-based clusters. Finally, it's familiar C/Java like syntax makes it easy for developers to quickly understand language constructs. The diagram in Figure 4 demonstrates the 4 portions of the Swift stack which contains (1)A high level Swift Script, (2)A compiler, namely STC, to generate (3)An intermediate code and (4)Execution of the intermediate code. The most recent version of Swift, namely Swift/T, supports function calls.[7] This work also presents a novel API to allow interaction between GeMTC and Swift/T. By integrating the GeMTC middleware into the Swift/T stack Swift is capable of calling C wrapper functions to CUDA kernels/applications directly.

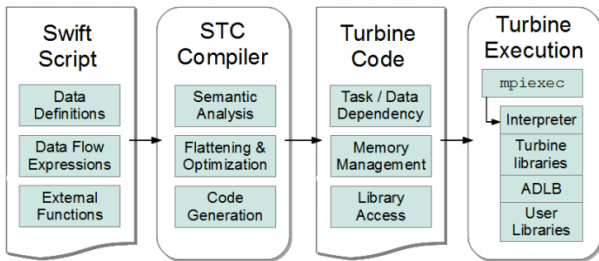


Fig. 4. Overview of the Swift/T tool chain.

IV. PRELIMINARY RESULTS

In this section we present preliminary results from the integration of GeMTC and Swift/T. In Figure 5 we demonstrate the efficiency of Swift/T and GeMTC on 4 Cray XK7 nodes. Swift/T is launching 50k system wide tasks for which there are known completion times. Efficiency is set to actual run time / expected runtime. Evaluation takes place on 1 to 4 nodes of

Cray XK7 machines with 156 GPU workers (the maximum) on each NVIDIA K20 GPU. Tasks lasting longer than 400ms are shown to have good efficiency and at 4-node scale we are capable of maintaining 4k tasks per second system wide.

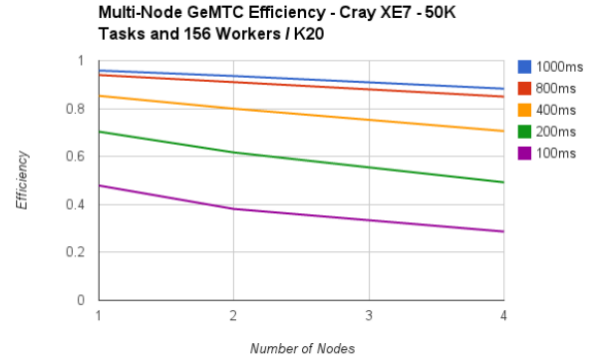


Fig. 5. GeMTC + Swift/T efficiency on 4 XK7 nodes.

V. CONCLUSIONS AND FUTURE WORK

In this work we designed and implemented the GeMTC framework. A sub-allocator provides improved memory management for dynamic tasks. Integration between GeMTC + Swift/T provides application support, data-flow parallelism, and multi-node scalability. Finally, we evaluated synthetic benchmarks and demonstrated improved performance. Future work aims to abstract for additional accelerator support, evaluate real applications such as the Open Protein Simulator (OOPS) [8], and improve current framework performance.

REFERENCES

- [1] S. J. Krieder and I. Raicu, "An overview of current and future computing accelerator architectures," 1st Greater Chicago Area System Research Workshop Poster Session, 2012.
- [2] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 22.
- [3] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Services, 2007 IEEE Congress on*. IEEE, 2007, pp. 199–206.
- [4] S. J. Krieder and I. Raicu, "Towards the support for many-task computing on many-core computing platforms," Doctoral Showcase, IEEE/ACM Supercomputing/SC, 2012.
- [5] S. J. Krieder, B. Grimmer, and I. Raicu, "Early experiences in running many-task computing workloads on gpgpus," XSEDE Poster Session, 2012.
- [6] NVIDIA, "Nvidia kepler_gk110 architecture whitepaper," <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, 2012.
- [7] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory data flow processing," in *Proc. CCGrid*, vol. 13.
- [8] A. N. Adhikari, J. Peng, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Modeling large regions in proteins: Applications to loops, termini, and folding," *Protein Science*, vol. 21, no. 1, pp. 107–121, 2012.