Exploiting Heterogeneous Parallel Accelerators to Improve Performance in Graph Analytics

Masab Ahmad and Omer Khan
University of Connecticut, Storrs, CT, USA
{masab.ahmad, khan}@uconn.edu

ABSTRACT

With the ever-increasing amount of data and input variations, portable performance is becoming harder to exploit on today's architectures. Computational setups generally utilize single architecture types such as either GPUs or large scale multicores for graph analytics. Disparities occur in performance and energy when a target algorithm and input benefits more from a different type of underlying parallel accelerator architecture than the one being used. Some algorithm-input combinations might perform more efficiently when utilizing a GPUs higher concurrency and bandwidth, while others may perform well using a multicore's stronger single-threaded performance and cache coherence hardware. This work aims to bridge this disparity by proposing an adaptive scheduler that learns these concurrency variations to select an optimal underlying accelerator, as well as the correct parallelization and concurrency choice within the accelerator architecture. Results show that scheduling on an optimal machine using the right concurrency choices provides significant performance and energy benefits on a variety of architectures executing graph analytics.

1. INTRODUCTION

Target applications utilizing graph, machine learning, and database processing in various ways have risen rapidly over the past decade. Their inputs are now processed in a plethora of architectures, ranging from miniature field mobile machines [1], to humongous supercomputers [2]. However big-data processing has almost never remained friendly towards underlying architectures, as bottlenecks remain from synchronization, memory access, and input dependent graph variations. Workload and input changes in such operational HPC setups render performance overheads due to unpredictable architectural variations.

However, big-data analytics can exploit such variations to a certain degree if provided with heterogeneous computing platforms [3]. Applications with little exploitable parallelism such as certain path planning graph workloads (e.g. A*), perform better on machine with better single threaded performance (such as Intel's Xeon Phi accelerators) [4]. On the other hand, other path planning graph workloads such as the Bellman-Ford algorithm with greater degrees of complexity and work, can be highly parallelized on concurrent machines, such as GPUs [5]. This disparity in choosing an optimal underlying architecture leads to performance, energy, and real-time bottlenecks in various computational paradigms ranging from embedded platforms to datacenters [6].

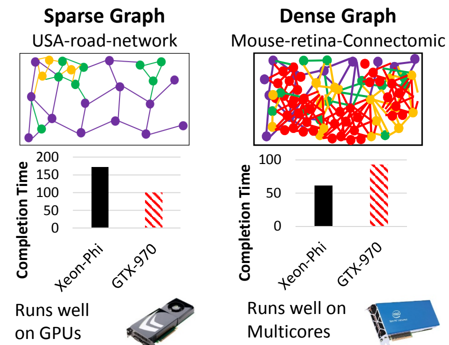

Figure 1: How input graph variations exhibit different performance across machine setups. Example shown for an optimized PageRank implementation running on an Intel 7120P Xeon Phi and an Nvidia GTX-970 GPU. One architecture does not optimize across both inputs.

To further solidify this disparity, an example is shown to show performance variations across two different architectures due to input changes. Figure 1 shows an OpenTuner-optimized PageRank implementation running two different input graphs on an Intel Xeon Phi 7120P multicore and an Nvidia GTX-970 GPU. Sparse graphs have less connectivity, and hence cause less stress on the memory subsystem [7]. Moreover as they require almost no synchronization, higher concurrency can be exploited, which is plentifully available on GPUs (which perform 50% better than the multicore in this case). Dense graphs on the other hand have higher connectivity between vertices that stresses the memory subsystem, thus adding more requirements for synchronization and single-threaded performance [8]. The Intel Xeon Phi multicore better exploits these characteristics, and performs ∼40% better than the GPU counterpart. Thus, for the same algorithm running two different input graphs causes performance variations on a variety of architectures [9].

Taking the example of graph algorithms, they already pose significant performance challenges due to the large amount of choices that can be availed to execute algorithm-input combinations [9, 10, 11]. This is further exacerbated by the

underlying architecture, where exploitable parallelism, synchronization requirements, and memory access variations, all add to execution complexity. Such variations, including the example shown above, mainly stem from input graph variations, where graph characteristics contribute to changes in optimal algorithmic-architectural deployment [12].

Traditional computing models utilizing primarily unary architecture types within a compute node (e.g. GPUs, or large scale multicores such as Intel Xeon Phis), typically fail to provide ample performance and energy benefits [13, 14]. GPU architectures have higher threading, weaker cache capacities, and higher bandwidth, which allows for faster processing on sparse graphs [15]. Large multicores, such as Intel Xeon Phis, expose less threading, however they have larger and better caching to exploit increased memory access and synchronization requirements in dense graphs. This tradeoffs make the case for having multiple parallel accelerator architectures integrated in computational setups executing graph analytic problems.

Future computational nodes are expected to have various heterogeneity architectures tightly coupled and connected via high speed interconnects. Prior works mainly involve operating system related runtimes such as Rinnegan [16], to improve resource utilization in single machine environments. This work extends these works to justify how various architectural and algorithmic choices and variations can be exploited to improve performance, energy, and other parameters in graph analytics. Challenges include 1) Efficiently learning to schedule a target algorithm-input combination onto its optimal architecture. 2) Heterogeneity across various algorithms, inputs, and underlying architectures. 3) Evaluating queries in a near real-time flow.

2. CONCURRENCY VARIATIONS IN HYBRID COMPUTATIONAL SETUPS

2.1 Architectural Variations within Parallel Accelerators

Input dependence greatly leads to variations in performance, stemming from the underlying architecture. This has much to do with available concurrency within a parallel accelerator. In terms of threading, this means that unique inputs have unique threading characteristics, such as large inputs performing well with increased threading. However, this scalability depends on available memory, if more threads request data from off-chip resources then memory access penalties can cause performance losses. Such variations can thus be looked at for various accelerators.

Threading capabilities are one of the many architectural variations that can be exploited within an accelerator. For example, in GPUs threading can be done on a massive scale. Local threads in blocks and warps are scheduled for cores and memory in a closer proximity. Similar analogy can be made for multi-threading in multicores. Larger amounts of local threads can exploit data reuse more efficiently, while smaller amounts can reduce traffic on the already small local caches in a GPU. More expansive global threading can exploit last-level caches better, as well as more availability of cores. However, it can also hinder synchronization and data movement costs if a target application *executes* better with smaller and tightly coupled threading.

Consequently all these architectural aspects add to choices that need to be viewed as a whole to optimize for benchamrks and inputs. A positive aspect of these set of choices is that they are known at static time (e.g. DRAM size, cache capacities, and available core counts), which is useful in scheduling within different accelerator machines [17].

2.2 Architectural Variations across Accelerators

Until now we have explained how choices exist within various machines. However, variations also occur *across* machines, such as in heterogeneous setups in use nowadays. E.g. CPUs have become the central brains in a system, running the OS and user support services, whereas GPUs are becoming more and more suited for big data type processing. With various accelerators in a given system, inputs and workloads scale differently across different accelerator setups. The shortest path problem described earlier performs better on GPUs for certain inputs, and on multicores for other inputs. Such choice exist for many other input–benchmark combinations for graph analytics.

Choices primarily occur in threading, available memory and memory bandwidth, and energy usage across accelerators. Large scale multicores such as Intel Xeon Phi has several tens of cores on-chip, with significant memory, memory bandwidth, and cache capacities. Moreover their stronger floating point unit capabilities allow them to perform better on workloads having floating point operations and larger working set. GPUs on the other hand have smaller floating point units and caches per core, and thus do not fare well for such inputs and workloads. However, they do perform really well on algorithms suitable for higher concurrency, i.e. more threads.

2.3 Scheduling Issues in Multi-Machine setups

Due to large variations in accelerator architectures, various issues arise when optimizing for near real-time scheduling. One of these issues is the variation in bandwidth, to local DDR memories within accelerators, as well as on the PCI bus connections. Some accelerators may have higher bandwidth, and thus may perform well on memory bound applications. Available core and thread counts also induces variations. Which accelerator should a particular input be executed on to achieve optimal performance? Accelerator cores optimized for floating-point operations on a large scale are expected to perform well on certain workloads with floating precision requirements such as PageRank and Community Detection. Thus, some problem-algorithm-implementation-input combinations may scale well on one accelerator versus another. Such choices need to be catered for in a scheduler that addresses this issue. Learning is therefore required for concurrency variations across accelerators.

3. EXPLOITING CONCURRENCY VARIATIONS ACROSS MULTIPLE CONNECTED ACCELERATORS

Due to input graph changes, variations in *problem-algorithm-implementation-input-machine* combinations need to be exploited for performance or other objectives. The relationships

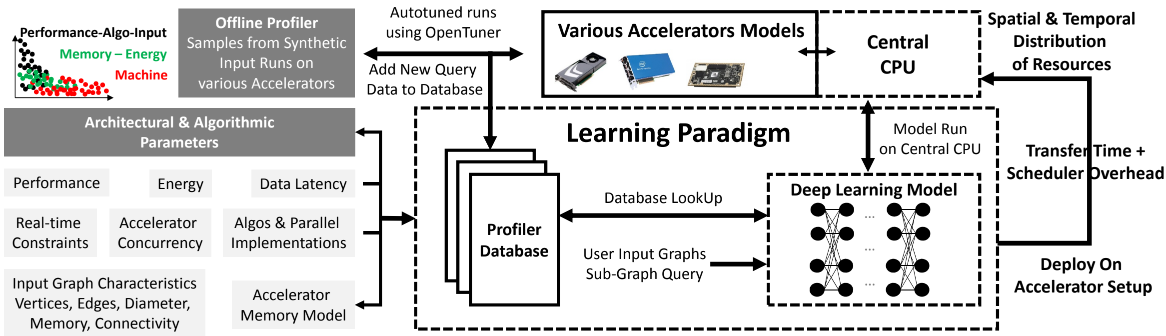

Figure 2: Scheduler Framework for Graph Analytics.

between algorithmic and architectural choices are already known to be highly non-linear, and thus using an analytical model or a linear equation system is undesirable. Many objectives therefore are required to be considered when creating such a framework. We formulate this target framework as a machine learning problem, with off-line learning requirements to exploit real-time opportunities. Figure 2 illustrates this framework in detail. The framework considers several architectural choices, which correspond to accelerator concurrency, memory, and latency models. Some algorithmic aspects are also considered, namely graph characteristics and some algorithmic choices stemming from various parallelization strategies for a chosen problem-algorithm combination. Additional characteristics such as energy and latency aspects are also considered.

The scheduler starts with a central machine learning paradigm, utilizing offline learning, and online evaluations. Offline learning is done on synthetic graph data, resembling how real data would occur in an online input stream. Combinations are created for various *problem-algorithm-implementation-input-machine*, and their performance and additional objective results are stored in an off-line database. These performance results are highly optimized using auto-tuning (OpenTuner used in this case), which create the database with completely optimized results. A machine learning model uses the database as training data to simply lookup something similar within the synthetic results. This part of training creates the biases and weights associated with the neural network learner.

Once the learner is fully trained, users can input real-world graphs to the learner. The learner then looks at input graph characteristics, such as size and sparsity, and determines which machine to optimally deploy it on. As all input parameters are coupled together, the optimal parameters (algorithmic and architectural) within a given machine, are also selected with the machine selection. The scheduler then deploys the target tuple configuration on an optimal architectural setup.

4. METHODOLOGY

Graph workloads are taken from the CRONO [18], Rodinia [3], and Pannotia [5] benchmark suites. Input graphs are taken from various sources including the SNAP repository. These graphs vary in sparsity and size, and take memory ranging from a few MB to several 10s of GB.

Two primary machines are used to emulate a heterogeneous multi-machine setup. A Xeon Phi 7120P multicore, having

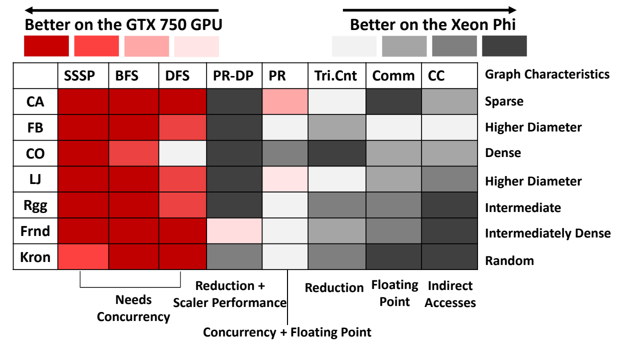

Figure 3: Inter-Machine Dependencies for various combinations. Reasons for variations also shown.

61 cores, 244 threads, and large caches and memory, and a GTX 750Ti GPU, that has 640 cores, and smaller caching and memory compared to the Phi.

To train the learner framework, synthetic input graphs are applied to the target algorithms and machines, which results in a training dataset of various tuple combinations. Training graphs are taken as uniform random [19], as well as Kronecker graphs [20]. These graphs model real-world graphs with significant similarity, and are expected to be ample for training. A 2.5*ms* overhead is added for the scheduler during the evaluation phase when real graph inputs are executed with various graph algorithms.

5. INITIAL RESULTS

To understand how different choices occur in multi-machine setups, various benchmark-input combinations are evaluated. Figure 3 shows how different input–benchmark combinations perform differently across the two target machines (the GPU and the Xeon Phi). Benchmarks with higher iteration requirements and longer outer loops, such as SSSP and BFS perform well with GPUs due to higher available thread counts. On the other hand, benchmarks such as Conn. Comp. (CC) and Triangle Counting that require indirect memory access and scalar computations perform well on the Xeon Phi. Input dependence is also plentifully present. DFS-CO and PR-CO perform well on the Phi, primarily due to inner loop parallelization across the edges which the Xeon Phi can better exploit because of larger local caches.

Figure 4 shows the results for all target graph benchmarks,

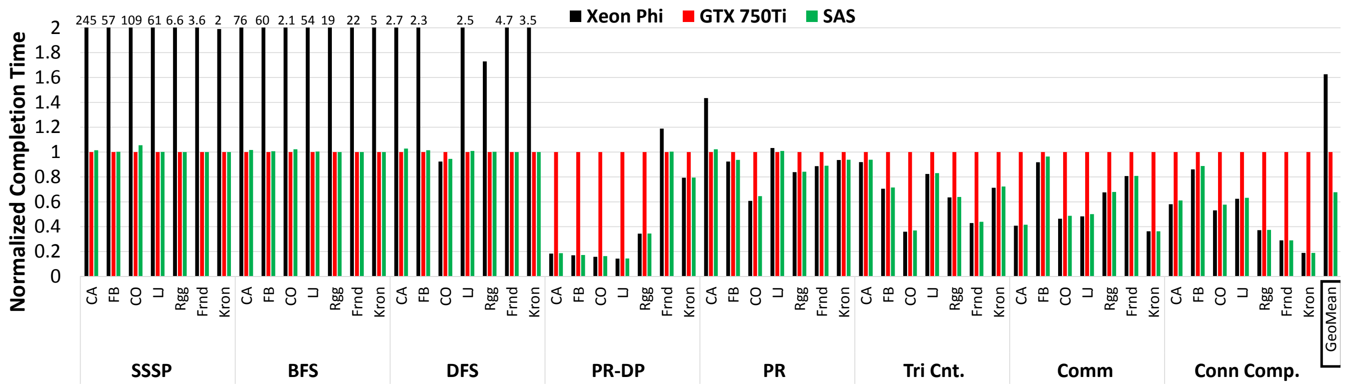

Figure 4: Scheduler Comparisons for various Graph Workloads.

and results are normalized with respect to the GPU completion time of each input–benchmark combination. Overall it can be seen that one machine does not perform well for all benchmarks. Benchmarks such as SSSP and BFS that have dependencies across iterations and require more parallelism fare better on the GPU. Other workloads, such as Community and PageRank perform better on the Xeon Phi due to more floating point requirements. Overall, the proposed learner schedules the optimal architectural choices within and across machines, which results in a average of 32% improvement over a GPU-only implementation, and a 58% improvement over a Phi-only implementation.

6. CONCLUSION

This paper shows that choices exist in graph applications within and across many-core machine implementations. This is because input-sensitivity in graph benchmarks leads to performance bottlenecks due to concurrency variations. However, the right selection of these architectural choices can be done in a near real-time setting that leads to a significant improvement in performance.

7. REFERENCES

[1] S. Iqbal, Y. Liang, and H. Grahn, "Parmibench - an open-source benchmark for embedded multiprocessor systems," *Computer Architecture Letters*, vol. 9, no. 2, pp. 45–48, Feb 2010.

[2] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 440–451.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[4] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "Unlocking ordered parallelism with the swarm architecture," *IEEE Micro*, vol. 36, no. 3, pp. 105–117, May 2016.

[5] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IEEE Int. Symph. on Workload Characterization (IISWC)*, Sept 2013.

[6] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. of the 2014 ACM SIG. Int. Conf. on Management of Data (SIGMOD)*. NY, USA: ACM, 2014.

[7] J. Leskovec and et al, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[8] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," in *Nature Neuroscience 17*, Sept 2014.

[9] M. Ahmad and O. Khan, "Gpu concurrency choices in graph analytics," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.

[10] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316.

[11] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38–49.

[12] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 379–390.

[13] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 528–540.

[14] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM Int. Symph. on*, May 2015.

[15] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014, pp. 140–149.

[16] S. Panneerselvam and M. Swift, "Rinnegan: Efficient resource use in heterogeneous architectures," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 373–386.

[17] M. Ahmad, C. J. Michael, and O. Khan, "Efficient situational scheduling of graph workloads on single-chip multicores and gpus," *IEEE Micro*, vol. 37, no. 1, pp. 30–40, Jan 2017.

[18] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono : A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proc. of IEEE Int. Symposium on Workload Characterization*, ser. IISWC, 2015.

[19] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," 2006.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.