HeteroMap: A Runtime Performance Predictor for Efficient Processing of Graph Analytics on Heterogeneous Multi-Accelerators

Masab Ahmad[1], Halit Dogan[1], Christopher J. Michael[2], Omer Khan[1]
University of Connecticut, Storrs, CT, USA[1]
Naval Research Laboratory (NRL), John C. Stennis Space Center, MS, USA[2]

***Abstract*—With the ever-increasing amount of data and input variations, portable performance is becoming harder to exploit on today's architectures. Computational setups utilize single-chip processors, such as GPUs or large-scale multicores for graph analytics. Some algorithm-input combinations perform more efficiently when utilizing a GPU's higher concurrency and bandwidth, while others perform better with a multicore's stronger data caching capabilities. Architectural choices also occur within selected accelerators, where variables such as threading and thread placement need to be decided for optimal performance. This paper proposes a performance predictor paradigm for a heterogeneous parallel architecture where multiple disparate accelerators are integrated in an operational high performance computing setup. The predictor aims to improve graph processing efficiency by exploiting the underlying concurrency variations within and across the heterogeneous integrated accelerators using graph benchmark and input characteristics. The evaluation shows that intelligent and real-time selection of near-optimal concurrency choices provides performance benefits ranging from 5% to 3.8×, and an energy benefit averaging around 2.4× over the traditional single-accelerator setup.**

I. Introduction

Target applications that utilize graph processing are rising in a plethora of architectures [1, 2]. Future HPC datacenters are expected to have heterogeneous connected accelerators, with Cray and NVidia already edging on similar ideas [3, 4]. It has been indicated in prior works that graph analytics pose limitations when executed on a single accelerator setup [5] [6]. Thus, this paper proposes a multi-accelerator setup to situationally adapt the graph problem and input to the right machine and its concurrency configurations. To understand this problem, consider the iterative Bellman-Ford algorithm and its variants finding shortest paths. Such a graph algorithm lends itself for data-parallel execution since it easily allows graph chunks to be accessed in parallel. Hence, such an algorithm performs well on a GPU, since it exploits massively available threading to exploit parallelism [7]. On the other hand, algorithms such as Triangle Counting are not as parallel, and comprise of reductions on vertices that result in complex data access patterns. These access patterns lead to increased data movement and synchronization requirements [8]. Multicores perform well in such cases as they incorporate caching capabilities for efficient data movement and thread synchronization. These variations solidify the *need for diverse types of accelerators in a setup executing graph analytic workloads.*

Taking this problem into context, this paper takes a heterogeneous architecture that constitutes both types of competitive multicore and GPU accelerators connected under their own discrete memories. This setup exposes concurrency choices to graph applications, thus catering for the missing throughput and reuse capabilities in GPUs and multicores. Performance variations occur not only due to changes in benchmark characteristics, but also input changes within a benchmark, as well as different mappings of graph analytic benchmark-input combinations on different accelerators. These choices do not exist in a single accelerator setup. Algorithmically, in the presence of expensive synchronization on shared-data or indirect memory accesses, GPUs cannot perform as well as multicores. Multicores possess hardware cache coherence and a complex cache hierarchy to exploit performance in such cases. In various cases, the massive throughput of the GPU, or the data reuse of the multicore needs to be constrained to reduce stress on the memory system and data communication. One way to manage this is to spawn less threads in the workload [7]. Thus, choices occur both *within* and *across* accelerators, for different benchmarks and inputs.

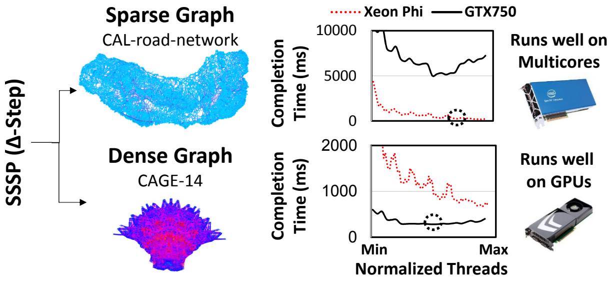


Fig. 1: How input graph variations exhibit different performance within and across underlying accelerators in SSSP.

Input dependence is known to play a big role in graph analytic performance [9]. An example of such a trade-off is shown in Figure 1, which shows an OpenTuner optimized [10] Δ-stepping single source shortest path (SSSP) algorithm [11] running a sparse and a dense graph on an Intel Xeon Phi 7120P multicore, and an Nvidia GTX-750TI GPU. Threads are varied from minimum total available threads to maximum total threads for both accelerators, and are normalized on the x-axis, while the y-axis shows completion time. The two accelerators are categorized as competitive as they possess similar compute capabilities.

The multicore performs better than the GPU for the sparse road network [12], as a higher graph diameter results in longer dependency chains that determine the optimal path between source and destination vertices. This linked traversal leads to more complex data access patterns that are more expensive on the GPU, as it does not possess the addressing capabilities to perform such complex data accesses. Moreover, the different phases in Δ-stepping result in more divergence and complex indirect addressing, which adds to GPU overheads. The multicore in this scenario performs several orders of magnitude faster than the GPU. The CAGE-14 graph [13] has a lower diameter, and thus requires less iterations to converge. Due to high density of edge connectivity, it lends itself to map optimally on a GPU. Larger available core and thread counts in GPU allow it to outperform the multicore by 3×. Even when the optimal accelerator is selected, there are a slew of machine choices within the accelerator to choose from. In the case of CAGE-14 graph, intermediate threading performs best on the GPU, as spawning more threads raises stress on the GPU's already small cache system. Machine choices within and across accelerators therefore need to be tuned based on different inputs to achieve optimal performance. Moreover, for different benchmarks, the patterns that lead to concurrency and data accesses also vary across graph analytics, which further motivates the need to tune this accelerator choice space.

This poses several questions: What patterns in graph benchmarks and inputs lead to best exploitation of concurrency within and across GPUs and multicores? What are the architectural differences in these machines that lend them for mapping to the diverse benchmark-input combinations? What are the run-time concurrency trade-offs of using one accelerator over another in a heterogeneous setup? Benchmarks and inputs reveal accelerator choices due to their direct correlations with the optimal architectural choices. Thus, graph benchmark and input choices need to be exposed systematically, after which a high level intelligent predictor tunes the accelerator choices. However, due to the increased high-dimensional space complexity and non-linear aspects of having multiple accelerators and their intra-concurrency choices, selecting the right choices becomes a hard problem.

This paper proposes a novel performance predictor framework, **HeteroMap**, which integrates benchmark and input choices to do dynamic selection of parameters within and across accelerators. The prediction framework captures program characteristics by intelligently discretizing graph benchmarks and inputs into easily expressible representative variables. Mappings of benchmark and input representations to inter- and intra-accelerator choices are done using a decision tree analytical model. The proposed analytical model is further automated using machine learning to amortize costs associated with the large graph algorithmic choice space. The automated model is trained using synthetically generated graph benchmarks [14, 15], and inputs [16, 17]. For a variety of graph analytic benchmarks executing real-world inputs, **HeteroMap** provides performance benefits ranging from 5% to 3.8× when compared to a single GPU-only or multicore-only setup.

II. Multi-Accelerator System

The target system utilizes discrete GPU and multicore accelerators. The setup considers either a weaker NVidia GTX-750Ti GPU or a stronger NVidia GTX-970 GPU, but not both at the same time. We also consider a weaker Intel Xeon Phi 7120P multicore or a stronger 40-core Intel Xeon E5-2650 v3 multicore. All multicore-GPU combination pairs are considered to analyze the inter- and intra-accelerator design space. This multi-accelerator system is used as a prototype to convey the underlying idea of mapping architectural choices using graph benchmarks and inputs. Figure 2 depicts an example multi-accelerator system showcasing a GPU and a Xeon Phi multicore with GDDR5 memories, as well as various architectural differences between associated accelerators. As memory size changes require architectural reconfigurations, evaluations are done on fixed memory sizes for each target accelerator. The design space of various combinations of memory sizes is also studied to analyze how main memory size changes affect performance in accelerators.

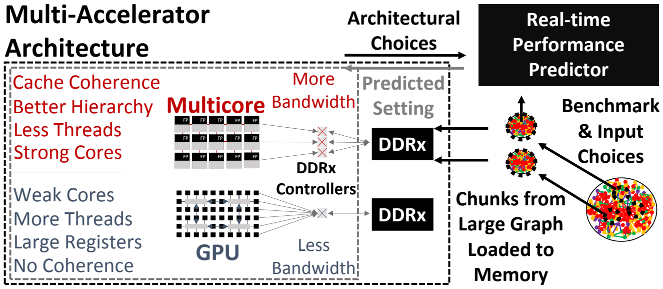


Fig. 2: Multi-accelerator system example with the run-time performance predictor for graph benchmarks and inputs.

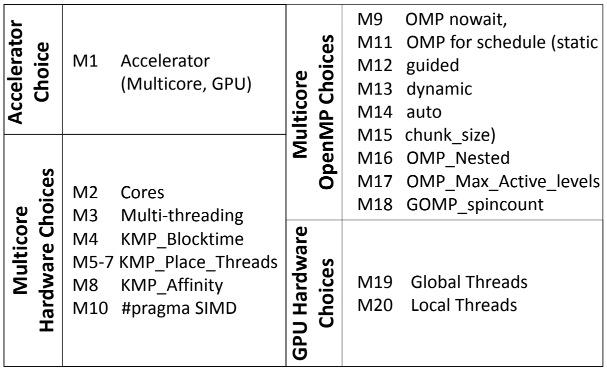
Input graph chunks are loaded in the accelerator's respective DDR memory for processing. The system is used in a way that graph benchmark-input combinations are loaded and executed with the appropriate architectural choices for individual accelerators with the mentioned discrete memory size constraint. In a real-time context, it is harder to allocate graph chunks and process them as larger graphs do not fit in main memory. Hence, chunks from larger graphs are thus extracted temporally using a state-of-the-art Stinger framework [18], and streamed in the accelerator's memory to be processed. This is similar to works in graph streaming [19]. The prediction paradigm takes in graph chunk characteristics, and predicts optimal architectural concurrency parameters for each chunk.

III. Performance Prediction Paradigm

Graph inputs consist of vertices, V, which are connected to other vertices via edges, E. Graph benchmarks loop around outer vertices and inner edges, and different phases in workloads have different complexities and have diverse data access patterns. Due to data access and synchronization pattern differences in graph inputs and workloads, different benchmarks and inputs perform optimally on different machines with different intra-accelerator settings. The multi-accelerator architecture in Figure 2 exposes these *intra- and inter- accelerator variations*, and we create a knowledge base from benchmarks and inputs that can be mapped to these machine choices.

A. Tuning the Intra- and Inter- Accelerator Choices

Various capabilities in GPU and multicore accelerators allow improved performance extraction for specific graph benchmark and input characteristics. This trade-off between accelerators is depicted as $M1$ in Figure 3, where either a GPU or a multicore can be selected. In GPUs, massively available threading hides data access latencies to deliver high throughput



Accelerator Choice	M1 Accelerator (Multicore, GPU)	Multicore OpenMP Choices	M9 OMP nowait, M11 OMP for schedule (static M12 guided M13 dynamic M14 auto M15 chunk_size) M16 OMP_Nested M17 OMP_Max_Active_levels M18 GOMP_spincount
Multicore Hardware Choices	M2 Cores M3 Multi-threading M4 KMP_Blocktime M5-7 KMP_Place_Threads M8 KMP_Affinity M10 #pragma SIMD	GPU Hardware Choices	M19 Global Threads M20 Local Threads

Fig. 3: Machine choices (M) for GPUs and multicores.

execution. This occurs in data-parallel workloads with small dependency chains and less shared data, and thus GPU accelerators must be selected for such cases. Although GPUs fare well with highly data-parallel execution, they under-perform when benchmarks have complex data access patterns, costly synchronization, and inter-thread data movement. Moreover, even in data-parallel workloads, the threading and throughput of a GPU may need to be constrained due to varying input sizes and densities to reduce stress on the memory system for optimal performance. This creates **two choices within a GPU**: `Global threading`, which distributes threads across the GPU chip, and `Local threading`, which specifies the thread count on a GPU core. These choices are listed in Figure 3 as GPU hardware choices, $M19-20$.

Multicores perform well for complex data access patterns by taking advantage of their cache reuse and cache coherence capabilities. Therefore, multicores should be selected if there is ample shared data. Multi-threading usage and placement intra-choices depend on the input graph characteristics such as edge density. Specifically for **multicore threading**, `KMP` *affinity/place_threads* are thread placement hardware choices in Figure 3, while `# pragma simd` controls SIMD usage. Thread placement may be compact or loose, and is important for data movement along with core and cache utilization. For example, threads may want to use cache slices of unused cores, which can be enabled by placing threads in the center of unused core clusters. This improves performance by reducing data movement and synchronization costs as threads are placed closer to the residing data. `KMP` *blocktime* is another parameter, which defines the time a thread waits before going to sleep. This is helpful during contention and load imbalance, as threads can go to sleep before polling on contended data.

Other parameters, such as those in the OpenMP paradigm, also have non-linear relationships with benchmarks and inputs, and are used to improve shared data reuse and movement costs. **Scheduling variables in OpenMP** involve dynamic scheduling, which control work distributions across parallel regions. Scheduling is controlled by `OMP for schedule`, which is tasked with *static, dynamic, guided, or auto* choices, and data tile/chunk sizes. Data scheduling is related to access patterns, which require dynamic scheduling on read-write shared data. This mitigates contention and data movement overheads [20]. Additional parameters such as `OMP_Nested` exploit nested parallelism within loops, while `OMP_Max_Active_Levels` states how many levels of parallelism can be nested. `GOMP_Spincount` defines how long threads actively wait for OpenMP calls. Larger times with this variable may be used to increase waiting times for threads if there is high contention. These OpenMP parameters are denoted as $M9,M11-18$, and are listed in Figure 3.

The M variable space is a function of the target benchmark and associated graph input, and this is the formulation required to achieve tuning of M parameters. All choices symbolize a non-linear mapping between benchmarks and graphs, and M choices. Thus, we create a benchmark and input graph representation space, denoted by B and I respectively. To minimize performance, a tuple vector, X, is constructed that takes benchmark choices $\vec{B}$, input choices $\vec{I}$, and accelerator choices $\vec{M}$, to minimize performance in the proposed architecture: $X(\vec{M}) = Min_{Perf}(\vec{B},\vec{I})$. The function, $Min_{Perf}()$ is the proposed configurator that finds M choices. To properly relate benchmark and inputs with M choices, B and I variables need to be extracted and classified for tuning. The next sections first describe B,I variables in the context of how they are expressed, and their relationships with machine choices.

B. *Input (I) Variables*

The most relevant input variables are graph size using vertex counts ($I1$) and edge density ($I2$), which specify the size of the graph and the density of computations. Higher graph sizes and densities can be divided into more threads, thus thread count selections in accelerators are directly correlated with $I1$ and $I2$. The maximum edge count of any vertex in the graph ($I3$) is also relevant as it defines how much deviation there is in edge connectivity from the average density using $I2$. This is used to define average per-thread work, as well as divergence in work between threads. Higher or lower per-thread work is used to decide how much local threading and/or SIMD to use, while work divergence is used to optimally place threads, in a selected accelerator. Graph diameter ($I4$) specifies the largest connectivity distance between any two vertices, specifying dependency chain sizes between vertices in a graph. $I4$ is obtained alongside input graphs or using runtime approximations [21]. This in turn expresses how much the memory system is going to be stressed during execution, as longer vertex dependency chains need to be remembered in memory. $I4$ is helpful in deciding which type of memory system needs to be tuned for an input graph.

All input variables are also easily expressible in percentages, as maximum vertex and edge count, maximum degree, and diameter, are known in literature [14]. These proposed I variables are used to classify a real input graphs to expose input variations, shown in Table I. These range from sparse road networks, social networks, to dense mouse brain graphs. **Input Graph Expression using I Variables:** I variables are deduced from graph data and are shown in Figure 4. These representations are simply obtained by normalizing the input graph's characteristic data, and setting it to a value between 0 and 1, with increments of 0.1, depending on the acquired value. I variables are normalized by comparing the input graph characteristics to the maximum values available in literature [25, 11] for these variables. Normalization is necessary,

TABLE I: Input Datasets.

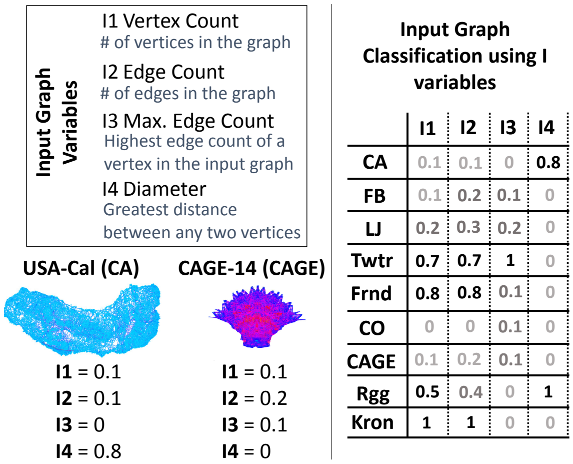
Evaluation Data	*#V*	*#E*	Max.Deg	Diameter
USA-Cal(CA)[12]	1.9M	4.7M	12	850
Facebook(FB)[22]	2.9M	41.9M	90K	12
Livejournal(LJ)	4.8M	85.7M	20K	16
Twiter(Twtr)[23]	41.7M	1.47B	3M	5
Friendster(Frnd)	65.6M	1.81B	5.2K	32
M. Ret. 3(CO)[24]	562	0.57M	1027	1
Cage14(CAGE)[13]	1.5M	25.6M	80	8
rgg-n-24(Rgg)[22]	16.8M	387M	40	2622
Kron.-Large(Kron)	134M	2.15B	16.0	12

as these characteristics need to be compared to each other to predict inter- and intra-accelerator choices. Furthermore, as graphs have extremely large variations among themselves in terms of characteristics, a logarithmic normalization is applied to further smoothen I values. Using the USA-Cal input graph as an example to compute I variables, vertex and edge counts in USA-Cal are low compared to the largest graphs such as Friendster. Hence $I1,2$ are set to 0.1 for USA-Cal, but 0.8 for Friendster. As the maximum degree of USA-Cal is also extremely low compared to the largest available degree in Twitter (which is 1), $I3$ is set as 0 in this case. However, its diameter is close to the highest available (850 is close to the largest diameter of 2622 for the Rgg graph). Therefore, we set $I4$ as 0.8 for USA-Cal and 1 for Twitter, and 0 for all other graphs. I variables for other input graphs are extracted similarly and shown in Figure 4.

C. Benchmark (B) Variables

In parallel graph algorithms, the outermost loop is parallelized, and traverses graph vertices in various phases such as highly parallel vertex division and pareto fronts, or less parallel reductions and push-pop phases. An algorithm may consist of multiple phases, where phases are separated by global thread barriers. Inner loops traverse edges, where data is addressed either directly using loop indexes, or indirectly using complex pointers. Data may be either shared as read-only or as read-write, and may require atomic updates. Read-write shared data may require local computations with either fixed point or floating point (FP) requirements to calculate output values to write into global data structures. These generic primitives are used to generate B benchmark variables. In this work, variables $B1-13$ define structural differences within graph-specific data structures and parallel phases, which are critical components in predicting machine choices.

Initial B variables are derived using outer loop parallel primitives. The outer loop may be data-parallel using Vertex division ($B1$), lending itself easily for execution with a larger number of independent threads. Pareto ($B2$) execution can also be applied on outer loops, where chunks of vertices mapped to threads statically increase with workload progression. These Pareto phases may also dynamically increase vertices in threads ($B3$). Graph workload phases may also take the form of Push-Pop $B4$ accesses, which add certain ordering constraints for processing. This in turn enforces dependencies, leading to complex data access patterns. Like Push-Pop accesses, Reductions ($B5$) contain more sequential work than other phases, and involve synchronization primitives with atomic operations. ($B4$) and ($B5$) phase types complicate data access and parallelism, leading to thread divergence. Therefore, GPUs may under perform for such scheduling patterns [26]. However, ($B1-3$) lend themselves for high parallelism on the GPU. These variables are important, as they describe how much each phase constitutes a benchmark. These **vertex processing and scheduling variables** ($B1-5$) are mutually exclusive, as programs are divided into phases. For example, a program may consist of 80% vertex division, and a 20% reduction phase. A programmer sets these variables by finding out how much a phase constitutes each benchmark.



	I1	I2	I3	I4
CA	0.1	0.1	0	0.8
FB	0.1	0.2	0.1	0
LJ	0.2	0.3	0.2	0
Twtr	0.7	0.7	1	0
Frnd	0.8	0.8	0.1	0
CO	0	0	0.1	0
CAGE	0.1	0.2	0.1	0
Rgg	0.5	0.4	0	1
Kron	1	1	0	0

Fig. 4: Input (I) model variables. Real graphs from Table I are discretized in increments of 0.1 from 0 - 1.

Compute type within phases may be FP computations done by the inner loops of workload phases. These FP computations determine if dedicated hardware units need to be exploited. This is shown by how much program data is specified as FP ($B6$), which trades-off accelerators, as some accelerators may have more FP capabilities than others. For example, if 20% of program data requires FP, then ($B6$) is set as 0.2. FP operations perform optimally on multicores if they are in a dense format to exploit SIMD capabilities. Therefore, knowing how much FP computations are needed can decide in mapping a benchmark to either a multicore or a GPU.

In terms of **memory access patterns**, addressing is either done with loop variables ($B7$), or by complex indirect addressing such as double pointers ($B8$). Complex addressing primitives are better handled in multicores as they possess larger caches to hold addressing metadata, and have faster ways of resolving complex pointers and addressing. Indirectly accessing and reusing data via addressing in the cache does not fare well with GPUs as they do not have the capabilities or enough cache sizes to hold such contents. A programmer sets ($B7,8$) by viewing what percentage of data is accessed indirectly, or by using loop indexes.

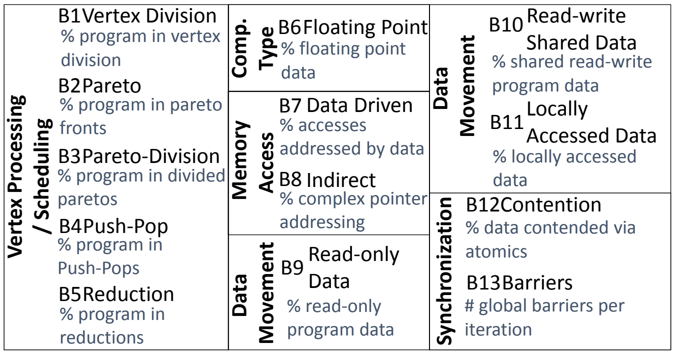
Runtime **data movement** is also diverse, and takes the form of read-only shared data ($B9$), read-write shared data ($B10$), and locally accessed data ($B11$). ($B9-10$) fare well on multicores as they have cache management mechanisms for efficient data movement between cores. ($B11$) is data that is locally operated in thread registers, where it depends on the accelerator's cores on how fast they process local computations. Each of these variables is expressed by the programmer as a percentage of the total accessed data.

Shared data may also require updates with **synchronization** ($B12$), where ($B12$) is viewed as the percentage of data requiring locks, as certain accelerators may have better performing atomics than others. The number of barriers in a workload separating phases ($B13$) also causes variations. If there are more locks and barriers in a benchmark, then it produces more opportunities for inter-thread communication to cause bottlenecks and load imbalance. ($B13$) is specified as the number of barriers between phases, and each barrier increments ($B13$) by 0.1, per iteration.

($B1-5$) are considered as independent variables. Although the interactions of remaining B variables are complex, these variables are not considered mutually exclusive. All benchmark variables are also easily expressible in percentages. The programmer specifies which B variables are interesting in a given benchmark. For simplicity, this section first uses a ✓representation to signify whether each B variable is specified or not in a benchmark. This classification is shown in the subsequent subsection.

Benchmark Expression using B Variables: Now that benchmark variables are defined, these variables can be used to classify real graph workloads. Graph workloads are thus acquired from a variety of benchmark suites, further specified in Section VI-B. These benchmarks are also listed in Figure 5, with the ✓representations showing if a B variable is used in a benchmark. Based on compile-time information about loops and inputs, loop indexes and data structure sizes are inferred, and are used to approximate relative strengths of B variables. As multicore and GPU versions of benchmarks use the same algorithms, their B variable classification remains the same.

Taking the case of SSSP-BF as an example, the only parallelization applied is vertex division, which enables $B1$ to be set as ✓. If the SSSP-Delta workload is used then parallel buckets are used to push and pop edges, setting $B4$ as ✓. The GAP version also uses a reduction to select a bucket to use in subsequent iterations, which sets $B5$ as ✓. In terms of program phases, the general distribution is that workloads use data-parallel vertex division $B1$ along with reductions $B5$. BFS uses only Pareto-division $B3$, and DFS uses only Push-Pop $B4$, as workload phases only contain one phase of these types. All workloads have data-driven accesses $B7$, and read-write shared data $B10$. DFS and Conn. Comp. have complex indirect data accesses, which are due to queuing and data-manipulated addressing, and these set $B8$ to ✓.



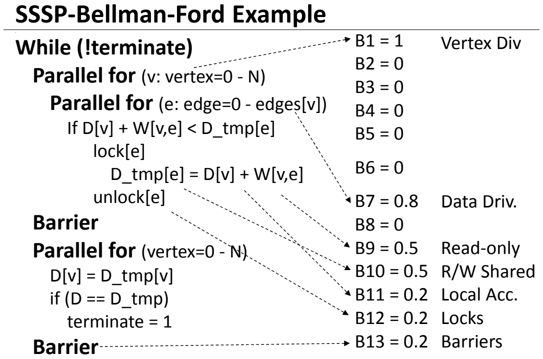
Benchmark Representation using B Variables

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13
SSSP-BF	✓						✓		✓	✓		✓	✓
SSSP-Delta	✓			✓	✓		✓			✓	✓	✓	✓
BFS			✓				✓		✓	✓		✓	✓
DFS				✓			✓	✓	✓	✓	✓		
PageRank-DP	✓				✓	✓	✓		✓	✓	✓		✓
PageRank	✓					✓	✓		✓	✓	✓		
Tri.Cnt.	✓				✓		✓		✓	✓		✓	✓
Comm.	✓				✓	✓	✓			✓	✓		✓
Conn. Comp.	✓	✓			✓		✓	✓	✓	✓	✓	✓	

Fig. 5: Benchmark (B) model variables. For simplicity, B variables that are used in a benchmark are discretized as ✓here for each graph benchmark utilized in this work.

B variables are percentages of program sections or percentages of data types used. These variables need to be normalized because simple ✓representations do not show intensities of each B variable. B variables are depicted within a range of 0 and 1, with increments of 0.1. Finer increments may be applied, however we keep the model simple by not using very fine increments. As graph workloads consist of only phases separated by barriers, values for $B1-5$ variables for phases add to 1 for all benchmarks. To assign values for more than one $B1-5$ variables in a benchmark (e.g. a workload having both Push-Pop and Reduction phases), the programmer decides approximately how much % code is in each phase. The programmer can statically view data structures to assign how much % of the structures fall in each of the remaining variable $B6-12$ categories. By specifying B variable values between 0 and 1, the programmer assigns percentages to variables, therefore properly assigning benchmark characteristics.

As an example, we take SSSP-BF to show this discretization, with its pseudocode shown in Figure 6 to visualize B variables. As all of the program code in SSSP-BF only uses vertex division to parallelize outer loops, thus $B1$ is set as 1 in Figure 6, while the remaining $B2-5$ are set as 0. $B6$ is set as 0 because SSSP-BF does not utilize FP operations. Most of the data accesses are done using loop indexes, such as accesses for *D_tmp[]*, *D[]*, and *W[]* arrays, therefore setting $B7$ to 0.8. $B8$ is set to 0 as there are no indirect accesses. Approximately half of the program data is composed of the input graph *W[]*, which is read-only by all threads. The other half are the distance arrays (*D_tmp[]* and *D[]*), which are read and written frequently by all threads. This sets $B9$ and $B10$ to 0.5 each. Local computations are done on *D_tmp[]*, which constitutes approximately 20% of program data, hence this sets $B11$ to 0.2. Locks are also applied only on the *D[]* array, which is half the size of the two distance arrays combined, and there are two barrier calls in the benchmark. This sets $B12$ and $B13$ to 0.2. Now that B, I variables are set, relationships between B, I and M variables can be found to predict M variables.



```
While (!terminate)
  Parallel for (v: vertex=0 - N)
    Parallel for (e: edge=0 - edges[v])
      If D[v] + W[v,e] < D_tmp[e]
        lock[e]
          D_tmp[e] = D[v] + W[v,e]
        unlock[e]
  Barrier
  Parallel for (vertex=0 - N)
    D[v] = D_tmp[v]
    if (D == D_tmp)
      terminate = 1
  Barrier
```

Fig. 6: Discretization of (B) variables for SSSP-Bellman-Ford (SSSP-BF) with increments of 0.1 from 0 - 1.

IV. HeteroMap Decision Tree Model

Patterns of M mappings allow visualization of accelerator choices with B and I variables. For example, benchmarks utilizing Push-Pop ($B4$) phases are expected to perform better on multicores than on GPUs. This is due to better data movement capabilities in multicore cache hierarchies, as well as better core performance for queuing operations. On the other hand, benchmarks with high data-level parallelism and local thread computations are expected to perform well on a GPU. This is because GPUs possess more threads to exploit available parallelism, and large register files to hold local computations. These relationships between B,I and M variables are used to create a simplified analytical decision tree model.

This section proposes a decision tree heuristic that analytically minimizes the choice space problem for performance (and energy if needed). Decision trees are easily readable and tunable, and thus allow for manual modeling. This model is expressed as an inter-accelerator model to first select an optimal accelerator, and then an intra-accelerator model to select concurrency choices within the accelerator. However, with 13 B variables, 4 I variables, and 20 M variables, the resulting choice space consists of thousands of combinations to select from. Hence, to simplify the prediction model, we only look at the most important variables that affect each M parameter. The complete M model is provided as a C/C++ program in the URL provided[1].

Inter-Accelerator ($M1$) Model: A 3-layer manually constructed decision tree is formulated, selecting an accelerator based on (B,I) combinations. As the complete decision tree is too large, we describe a few partial decision examples. For example, if a combination has $B1$ or $B2$ or $B3$ each with a value greater than 0.5, meaning it has lots of vertex level parallelism, then a GPU is chosen as it exploits this available parallelism. This allows workloads such as SSSP-BF and BFS to run on the GPU. On the other hand, if a benchmark has serial Push-Pop accesses ($B4$) with a high graph density, then the multicore is selected as it performs well on Push-Pop accesses with the dense graph fitting in its local caches. In another example, if a benchmark has a high value of $B5$ (reductions) with some FP ($B6$), and negligible local computations ($B11$), then the GPU is selected. This is because GPUs perform well with reductions having low local computations, meaning the small GPU threads can make fast progress using their small caches.

The multicore is selected for the case with reductions ($B5$) and read-write shared data ($B10$). This is because the cache capabilities in multicores allow faster operations on shared data, while synchronization primitives required for reductions on vertices also perform well due to faster inter-thread communication abilities. For large graphs with $I1 > 0.5$, benchmarks with indirect addressing are also run on the multicore for this reason. Larger graphs running with benchmarks requiring FP operations ($B6$) are also run on the multicore as it has a stronger memory hierarchy and FP capabilities. Thus, workloads such as Conn. Comp., PageRank, and Comm. are run on multicores if graphs are large.

A threshold of 0.5 is set as default to select between the GPU and the multicore as it shows the unbiased mid-point in normalized B,I values. For example, for high reduction and read-write shared data values ($B5 > 0.5$ and $B10 > 0.5$), multicores are selected. The execution model assumes that the programmer has to input such values, and hence selecting the mid-point seems to be the easiest way to acquire ample performance. Other thresholds may also work by fine tuning thresholds, however this is left as future work.

Intra-Accelerator ($M2-20$) Selection: Intra-accelerator calculations are more complicated due to non-linear $B-I$ to M relationships, solidifying the need to create a simpler linear equation model. Linear equations are of the form $y = ax + k$, which are converted to the following equation when input (B,I) and output M variables are linearized.

$$M = a(B,I) + k$$

As all M variables need to be set to a minimum value, k is used to specify this value. For example, when using a multicore, at least one core must be used, which sets $k = 1$ for variable $M2$. k values for other M variables are set similarly. The term $a(B,I)$ may incorporate linear relationships of different B,I variables. These relationships are intuitively derived using visualization of relationships between B,I and M variables. In some cases, an M variable may either be set or unset, and thus a threshold of 0.5 is used for such cases after resolving the equation result. Similar to $M1$, $B-I$ to M relationships for the rest of the M variables augment to many partial linear equations.

In GPUs, if the graph is dense (seen from I variables), then more local threads are desirable to parallelize edges, making GPU local threads ($M20$) proportional to the graph density. To obtain the deployable value, the acquired normalized result from the above mentioned relationship is multiplied with the maximum value of the machine variable being applied (GPU local threads in this case). This is given by the variable `CL_KERNEL_WORK_GROUP_SIZE` for OpenCL (simplified to *max_local_threads*). This relationship with the added constant (k=1 for GPU local threads as at least 1 thread must be spawned), is thus shown by the following equation:

$$M20 = Avg.Deg * max\_local\_threads + k$$

$$Avg.Deg = |I3 - (I2/I1)|$$

[1] HeteroMap Repository: https://github.com/masabahmad/HeteroMap

GPU global threads (M19) derive from $I1$, as outer loops are parallelized among threads. This implies that if there are more vertices, then more threads can be spawned for additional parallelism, resulting in the following relation:

$$M19 = I1 * max\_global\_threads + k$$

Similarly, in multicores, cores depend on the available parallelism in the outer loop, as more vertices can be parallelized among more cores with their additional cache slices. This is similar to the derivation of $M19$. Multi-threading/SIMD are also a function of the graph density, similar to $M20$. The higher the graph density, the larger the inner loops, meaning more threads or a wider SIMD per core must be spawned. These two variables are given by the following equations:

$$M2 = I1 * max\_cores + k$$

$$M3,10 = Avg.Deg * max\_multi - threading + k$$

The thread blocktime parameter ($M4$) defines thread wait times (max_thread_wait_time is set to be 1000ms, while the minimum can be set as 1ms). Threads are known to wait on locks and barriers via OS calls, and higher wait times are associated with higher contention levels. Thus, this parameter is acquired by taking the average of $B12$ and $B13$ as it depends on contention, and by setting $k = 1$, as shown by the following equation. The purpose of this equation is to correlate thread wait times to contention.

$$M4 = B12 + B13/2 * max\_thread\_wait\_time + k$$

In multicores, threads are placed in a more fine-grained manner, using variables $M5-7$. Thread placements not only depend on the average degree of the graph, but also on the graph diameter, as it determines temporal progression of work within a graph. Thread placement variables consist of three variables to create placement combinations: core ids ($M5$), thread ids ($M6$), and thread offsets ($M7$). Higher deviations between $I3$ and the average degree signifies variations in edge mapping across the chip. Thus, threads need to be placed loosely across the chip. A higher graph diameter depicts longer dependency chains between vertices, meaning that each thread needs to work longer to achieve desired outputs. This means that more threads are required, as vertices remain idle due to threads being busy waiting on longer dependencies. Thus, variables $M5-7$ are calculated by taking the average of average degree and the diameter:

$$M5-7 = Avg.Deg.Dia * max\_thread\_placement + k$$

$$Avg.Deg.Dia = |(I4 + Avg.Deg)/2|$$

Thread affinities in multicores mean pinning threads to cores in movable or strictly compact ways. Movable in this case means threads may be moved around by the OS or OpenMP scheduler if it determines that performance may be gained by moving threads to other cores. Again, affinity is related to thread placement, hence a relationship with $Avg.Deg.Dia$ is assumed. However, pinning threads to specific cores also relates to read-write shared data ($B10$), as performance improves when shared data is not moved between cores. In such cases, if $B10$ is high then threads need not be moved between cores to avoid unnecessary data movement. In the minimum case for k, all threads may be moved around by the scheduler, setting $k = 0$. Thus, thread affinity may be taken as the average of $Avg.Deg.Dia$ and $B10$, as shown by the following equation.

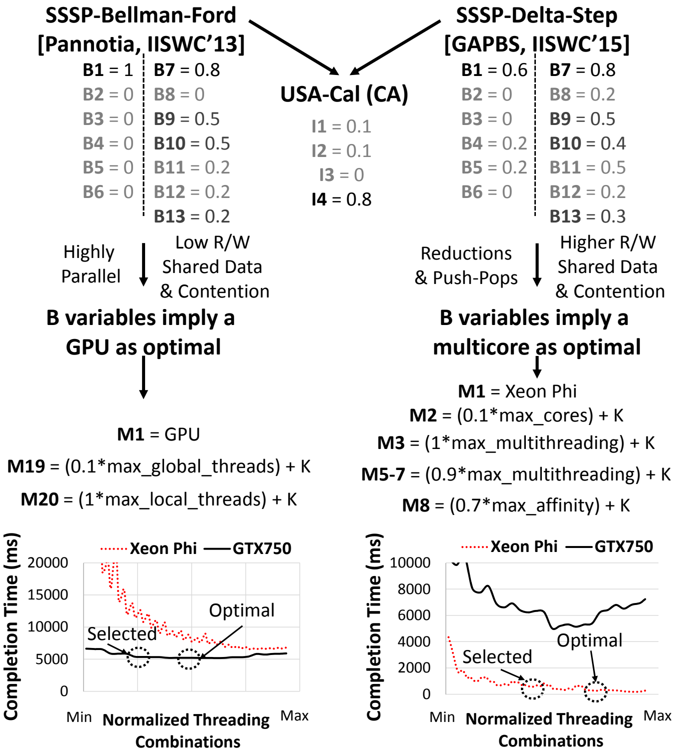


Fig. 7: Decision Tree Heuristic Model flow for SSSP-BF and SSSP-Delta with the USA-Cal input graph. The proposed model predicts and selects nine M choices.

$$M8 = Avg.Deg.Dia + B10/2 * max\_thread\_placement + k$$

If a calculated value resolves to a larger than maximum value for an M variable, then a ceiling function sets it to its maximum value. The proposed M variable equations are also expected to work for other GPUs and multicores, including CPU multicores. The remaining M parameters pertain to OpenMP choices not shown here due to space constraints, but are described in the HeteroMap repository[1].

M Choice Selection Example: In lieu of the shown relationships of B,I variables with M variables, we show an example of how M variables are predicted using the proposed model. Figure 7 shows this flow for SSSP-BF and SSSP-Delta running with the USA-Cal (CA) input graph. Discretized B variables are shown for the two benchmarks and the input graph, which are acquired by benchmark profiling. Using a visual inspection, B variables for SSSP-BF are more inclined towards exhibiting a highly parallel workload that has low read-write shared data and contention. This implies that SSSP-BF is expected to perform optimally on a GPU. On the other hand, SSSP-Delta has more sequential-like functions, such as reductions and the use of push-pop structures. This also makes its data more contended and shared in terms of reads and

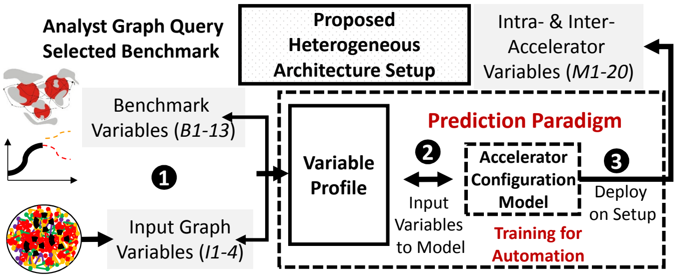

Analyst Graph Query
Selected Benchmark
Proposed Heterogeneous Architecture Setup
Intra- & Inter-Accelerator Variables (M1-20)
Benchmark Variables (B1-13)
❶
Input Graph Variables (I1-4)
Variable Profile
Prediction Paradigm
❷
Input Variables to Model
Accelerator Configuration Model
❸
Deploy on Setup
Training for Automation

Fig. 8: HeteroMap Framework Flow for the Multi-Accelerator Architecture.

writes. This implies that SSSP-Delta is expected to perform optimally on a multicore (Xeon Phi used in this case) using the proposed B variables.

Intra-accelerator M variables are predicted using the proposed equations. For SSSP-BF selecting the GPU case, $M19,20$ are calculated using the vertex count, $I1$, and the average degree respectively. These resolve to values of 0.1 for $M19$ and 1 for $M20$, meaning that only some global threading is required, but maximum local threading is to be deployed. For SSSP-Delta, $M1$ resolves to select the multicore. Furthermore, $M2$ and $M3$ selections follow $M19$ and $M20$ as the input graph retains the same $I1-3$ variables. This results in $M2$ resolving to 7 cores and $M3$ resolving to its maximum value of 4 threads per core. Thread placement variables, $M5-7$, resolve to 0.9 due the high indicated diameter in the CA graph, meaning that very loose thread placement is required. These calculated variables are then deployed, which results in a selected performance as shown in Figure 7.

To find the optimal performance point, all M variables are swept and completion times are acquired for each of the two benchmarks. Figure 7 shows these performance curves, along with selected and optimal performance points, where the selected threading results in about a 15% performance difference from the optimal case. This is because the decision tree heuristic and relationship equations do not take into account all the B and I variables for each M variable. Thus, some performance exploitation remains left out, as linearizations only help so much for non-linear relationships. Acquiring the optimal point is only possible if all B and I variables are tuned exhaustively for each M combination. This is not possible using a manual decision tree and linear equations, and thus M choice selections need to be automated with more complex models to reason about these relationships.

V. HeteroMap Framework Automation

This section formulates HeteroMap's predictors in an automated fashion, shown in Figure 8. Configuration starts with a central performance prediction paradigm, utilizing off-line learning, and real-time on-line evaluations. Several predictors are evaluated, namely deep learning and regression based predictors, that show trade-offs in terms of overhead, accuracy, and performance. A programmer first sets (B,I) variables for a particular benchmark and input ❶, after which the variable profile is input to the model (decision tree heuristic or automated) ❷. The predicted M parameters are then deployed on the heterogeneous accelerator setup ❸.

```
Generated Synthetic Examples                              B Values
Example 1
Parallel for (vertex=0 to N)  //vertex division           B1 = 1
   Parallel for (edge=0 to edges[v])                      B8 = 0.8
      Array[Array[edge]]  //Indirect Accesses             B9 = 0.9
        = Local_Computation (R/W work)                    B11 = 0.9
Example 2
Parallel for (vertex=0 to N)  //Pareto Division           B3 = 0.8
   Parallel for (edge=0 to edges[v])                      B5 = 0.2
      Lock[edge]                                          B6 = 0.5
         Array[edge] = Local_Computation (FP)             B11 = 0.8
      UnLock[edge]                                        B12 = 0.1
Barrier                                                   B13 = 0.1
Parallel for (vertex=0 to N)  //Reduction Phase
   Reduction(Array)
```

Fig. 9: Example synthetic benchmarks generated.

Offline Learning Formulation: As learning and evaluation cannot be done on the same benchmarks and inputs, synthetically generated mechanisms are required for off-line training. Synthetic variants are generated using formulations in existing benchmark tools [15, 27], and graph generators (Uniform random [16] and Kronecker [17]). These are well-known to represent real inputs [28], and are thus considered good contenders for training. For synthetic benchmarks, the formulation described in Section III generates various generic micro benchmarks. This generalization follows the $V-E$ formulation of graph loops, and these loops form phases in a benchmark, with each phase having unique characteristics such as read-write shared data or FP arithmetic. Phases are separated by barriers to propagate values to other threads.

Figure 9 shows how diverse synthetic benchmarks are created. Mixes of phases (varying $B1-5$ values) are obtained by having different $B1-5$ phases, along with loop variations such as read-write data, contention, and FP requirements (varying $B6-13$ values). This creates a large synthetic space, as $B1-5$ can create up to a hundred combinations, while variations of $B6-13$ create many more. Two generated examples are shown in Figure 9, with the first example having a vertex division phase writing local computations to shared data using indirect addressing. The second example shows two phases separated by barriers, with the first phase having pareto division updating a shared array using local computation via locks, and the second phase doing a reduction. B values are also shown for each of the examples (derived using Figure 5), and these are input to the learning model to be run as training data.

Training: Several million samples are generated from different B,I combinations mapping to M variables, which shows the *complexity of this space*. This stems from several thousand synthetic B,I-varying combinations. For a particular synthetic graph B,I combination, only one $\vec{M}$ combination tuple is selected, which provides the best performance, as a model would like to train on close to optimal parameters. The resulting training dataset thus has output architecture variable values that provide the best performance on synthetic benchmark-input combinations. In the case of the GTX-750 - Xeon Phi setup, training takes several hours if millions of combinations are run in parallel on each accelerator. These B,I combinations are run and their optimal M selections are stored in an off-line database for training. These performance results are highly optimized using auto-tuning (OpenTuner used in this case). This creates a *profiler* database of B,I,M tuples residing in the CPU file system, which is indexed using B,I tuples to get M solutions. Overheads with this training are performed per multi-accelerator setup, and are not included in evaluation, as training only needs to be done once per setup.

A. Online Evaluation

After training the automated model, HeteroMap takes in real benchmarks and graphs for evaluation. Benchmarks and inputs are first discretized into (B,I) variables, after which their M variables are predicted. This process is depicted in Figure 7. This work does not consider temporal aspects, where program parts are run on either accelerator. As only on-chip architectural characteristics of accelerators are compared to simplify complexity, memory transfer variations are not taken into account, and only the time spent in processing the graph on-chip is analyzed. However, we still do a sensitivity study in Section VII-D to show how HeteroMap responds to memory and architecture changes. Model training and database derivations are all done on a host CPU. Once all architectural choices are decided, HeteroMap deploys the benchmark-input combination on an accelerator. The overhead of HeteroMap during runtime evaluation phase is added to the overall completion time. Different predictors are analyzed for automation, namely the Deep Learning model, and the Regression model.

B. Deep Learning Prediction Model

Neural networks are known to effectively learn on non-linear characteristics, and may be efficiently re-trained for various configurations and programmer-driven strategies. Such networks learn on non-linear performance curves, which changes neuron weights and biases to create complex equation representations within the neural network path from inputs to outputs. Figure 10 shows the proposed neural network with 4 layers and 32 neurons per layer. Benchmark-input characteristics are characterized as 17 input neurons, with each neuron set for a benchmark and input variable. Similarly, output neurons are categorized for each M choice. Several works use the internal hidden neuron amount that is at least twice the size of the output neurons [29]. We thus take the internal neuron count as 128 (rounding off to the nearest power of 2) [30]. The network is configured as a feed-forward neural network and the size of the network is selected by balancing the trade-offs between learner complexity, accuracy, and overhead. Non-linear performance curves can also be captured using a regression model, as outlined next.

C. Regression Prediction Model

A non-linear regression (similar to [31]) is presented that finds the optimal choice configurations. Regression models are much simpler than neural models, as they need fewer equations. However, they do require higher orders and variable coefficients, which demand more multiplications, increasing complexity. These trade-offs may cause variations in deciding which learning model to use for optimal performance. This proposed regression model is fitted via Matlab, and then ported to C++ for performance comparisons. It is analyzed that a 7th order model fits well (provides an 85% accuracy for curve predictions) for the target choices. Models with lower order do not have sufficient classification accuracy, and models with higher orders have higher performance overheads.

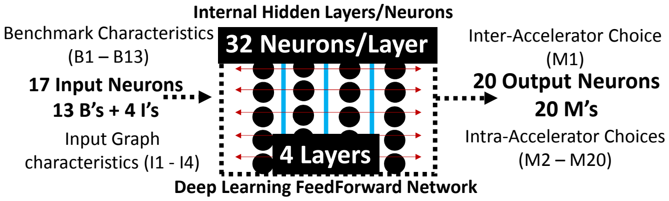

Internal Hidden Layers/Neurons
Benchmark Characteristics
(B1 – B13)
32 Neurons/Layer
Inter-Accelerator Choice
(M1)
17 Input Neurons
13 B's + 4 I's
20 Output Neurons
20 M's
Input Graph
characteristics (I1 - I4)
4 Layers
Intra-Accelerator Choices
(M2 – M20)
Deep Learning FeedForward Network

Fig. 10: Neural Network showing network parameters.

VI. Methodology

A. Accelerator Configurations

Two accelerators are primarily evaluated to build the multi-accelerator architecture, NVidia GTX-750TI and Intel Xeon Phi 7120P (parameters listed in Table II). These accelerators are competitive as their compute performance (single/double precision) overlap. Although the double precision capability of the Xeon Phi is higher, not all benchmark combinations require it during execution, and hence it contributes to the chip differences between accelerators which vary performance. The main memory used by both accelerators is pinned to the smallest one available. Memory size is not considered as a first-order effect in our work due to the fact that the whole architecture needs to be reconfigured and relearned for memory size changes. Still, a sensitivity study is done to show memory size effects, where the memories of both accelerators are swept and performance is acquired for all combinations. Storage to stand-alone memory transfer times are not measured, as they are assumed to be constant.

To evaluate with a more powerful GPU, we choose an NVidia GTX-970 to replace the smaller GPU for the multi-accelerator setup. GTX-970 incorporated 1664 cores with 3.5 TFLOPs single-precision and 0.1 TFLOPs double-precision compute capability, and has a larger 4 GB memory size. This work also evaluates an Intel Xeon E5-2650 v3 multicore having 10 hyper-threaded cores in 4 sockets, executing at 2.30GHz, with a 1TB DDR4 RAM. In addition to the primary (GTX-750TI, Xeon Phi) configuration, the following accelerator combinations are analyzed: (GTX-970, Xeon Phi), (GTX-750TI, CPU-40-Core), and (GTX-970, CPU-40-Core).

B. Benchmarks

For multicore benchmarks, SSSP-Bellman-Ford (SSSP-BF), BFS, DFS, PageRank, PageRank-DP, Triangle Counting (Tri.Cnt.), Community Detection (Comm.), and Connected Components (Conn. Comp.) are acquired from CRONO [8], MiBench [32], and Rodinia [33]. As SSSP-BF may not provide optimal performance on lower core counts in multicores, an SSSP implementation using Δ-Stepping (SSSP-Delta) is also acquired from the GAP benchmark suite [11] and compared.

TABLE II: Primary Accelerator Configuration.

	GTX-750Ti	**Xeon Phi 7120P**
Cores, Threads	640, Many	61, 244
Cache Size, Coherence	2MB, No	32MB, Yes
Mem. (GB), BW. (GB/s)	2, 86	2, 352
Single-Precision (TFlops)	1.3	2.4
Double-Precision (TFlops)	0.04	1.2

TABLE III: Synthetic Input Datasets.

Training Data	**#Vertices**	**#Edges**	**Avg.Deg.**	**Size(GB)**
Unif. Rand. [16]	16-65M	16-2B	1-32K	0.01-32
Kronecker [17]	16-65M	16-2B	1-32K	0.01-32

These versions use pthread/OpenMP implementations to run on multicores (using the offload programming model). For GPUs, benchmarks are acquired from Pannotia [5] and Rodinia [33] for OpenCL workloads, which provide SSSP, BFS, PageRank, and PageRank-DP. The remaining benchmarks are ported from the multicore implementations to OpenCL.

C. Processing Metrics

Original benchmarks from various benchmark suites are not optimized. In this case, they manually tuned as well to compare the proposed architecture and configuration framework to an ideal case. For fair comparison, the same algorithm within the benchmark is run on both accelerators. Training is therefore done to optimize all parameters offline using OpenTuner [10]. HeteroMap's output is compared with an ideal output that manually optimizes by running all possible configurations. Percentage accuracies are found by comparing the integer outputs (constituting choice selections) of the learners. Accuracy is measured by finding the percentage difference acquired performance using the proposed predictors, to the ideal case that optimizes all M variables. Target baselines are also taken from multicore-only and GPU-only runs. Different learners, namely regression and adaptive libraries, are also compared with HeteroMap in terms of accuracy and overheads. Completion times are compared for all benchmarks. Synthetically generated graphs for training the automated learners are depicted in Table III.

Graphs that are larger than the accelerator's main memory size are broken into chunks and processed one by one spatio-temporally using the Stinger framework [18]. To maintain fairness between accelerators, memory transfer times are not included in the completion time. Thus, only the time spent on the accelerator is measured, where the overhead of HeteroMap is added to the completion time. Energy numbers are also compared to allow the framework to utilize it as a metric. Power measurements are acquired using `micsmc` [34] and `powerstat` [35] utilities. Core utilization is measured using `nvprof` and `PAPI` [36, 37], and is the time each core spends in executing instructions in the pipeline.

VII. Evaluation

This section first evaluates HeteroMap by selecting an optimal learning model, and then compares to multicore-only and GPU-only baselines. Primary comparisons and analysis are done using the Xeon Phi and GTX-750Ti GPU setup. The various automated performance predictors are also compared with a baseline which optimizes all choices with no learner overheads (marked as ideal).

TABLE IV: Learning Model Strategies. Speedup shown over the GTX-750 GPU as it is the better baseline case.

Learner	**SpeedUp**	**Accuracy**	**Overhead**
	(%)	(%)	(ms)
Decision Tree	28	86.2	0.10
Linear Regression	6	50.1	0.05
Multi Regression	27	85.4	4.11
Adaptive Library[38]	8	56.5	0.17
Deep.16[26, 39]	11	59.3	1.52
Deep.32	22	68.4	2.52
Deep.64	26	82.2	3.01
Deep.128	**31**	**90.5**	**3.48**
Deep.256	30	92.9	6.39

A. Selecting a Learning Model

It is important to understand whether different learners in HeteroMap are optimal for the given choice space. We therefore take different parallel learning algorithms for comparison. Multiple Non-Linear Regression (fitted equations from Section V-C) and Decision Trees (IF-ELSE systems using thresholds from Section IV) are thus compared. A simple linear regression is also trained and compared. XAPP [31] uses regression with more than 7 variables, similar to the one evaluated in this paper. Rinnegan [38] uses a performance model adaptive library scheme, which profiles program performance and then uses a simple model equation to predict performance. The equation's output is directly proportional to only the data movement and accelerator utilization parameters given by a programmer/profiler. Deep learners are compared using various model sizes (explained in Section V-B). All are trained with the same amount of training data/time used for the proposed learners. Geomean completion times are taken for all benchmark-input combinations, and the speedup of each performance predictor is shown over the GPU.

Table IV shows that the adaptive library and linear regression paradigms do not perform well for our setup. This happens because of non-linear variations associated with graph benchmark-input combinations and multi-accelerator architecture choices. Regression does perform well enough, and results in a higher overhead, as complex equations are required to maintain accuracy. The decision tree model from Section IV provides low overhead, but does not provide a comparable speedup to the best deep learning model. Larger deep learners follow quadratic trends in overheads and classification accuracy. This raises the acquired speedup to a certain extent, after which returns diminish due to the increasing overhead. Overall, a speedup of 31% is acquired using the deep learning model as shown in Table IV, with a classification accuracy of 90.5% and overhead of 3.48*ms*. Hence, all further evaluations are done using the deep learning model with 128 neurons.

B. Performance Variations

In various cases either accelerator will be better in performance over the other. Figure 11 shows these variations for all

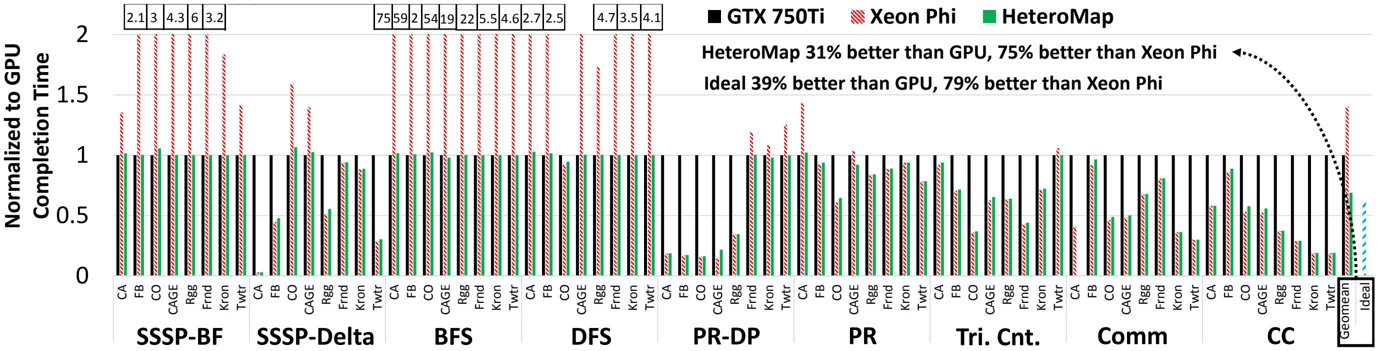


Fig. 11: Scheduler Comparisons for Graph Workload-Input Combinations (All results normalized to the GTX750Ti GPU implementation) (Higher is worse).

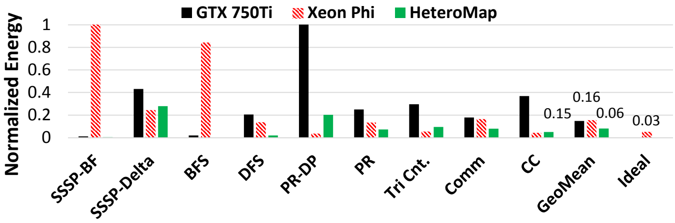


Fig. 12: Energy benefits averaged for various inputs for a given benchmark. (Xeon Phi vs. GTX 750Ti). All results normalized to the maximal energy used for any $B-I$ combination.

benchmark-input combinations with the deep learning model. The results include the framework's performance overhead in selecting a combination.

GPU-Biased Combinations: Benchmark-input combinations with highly concurrent algorithms, such as SSSP-BF, BFS, and DFS mostly fare well with the GPU. Their work division and parallelization strategy benefits from an excess of threads, which are available on the GPU. Due to the nature of their critical sections and data structures, the Xeon Phi cannot exploit its SIMD capabilities, and hence it performs poorly compared to a GPU. In the case of DFS-CO, the multicore outperform the GPU, as it uses additional inner loop parallelization. Such workloads are therefore easier for the learner to configure, as their performance curves remain biased towards the GPU.

Multicore-Biased Combinations: When benchmarks require FP capabilities they perform well on the multicore. Thus, PR, PR-DP, and COMM benchmarks perform well on the Xeon Phi as they require FP capabilities. When benchmarks require push-pop accesses on structures, alongside reductions (SSSP-Delta), then these benchmark-input combinations also perform well on the Xeon Phi. Some notable exceptions in these cases are Frnd. and Kron. graphs, which perform better on the GPU because they are large and require more threads. PR-CA does not perform well on a Xeon Phi, because it cannot take advantage of the SIMD capabilities due to the lack of density. The critical section in PR is also not large enough for the multicore to have any advantage, hence the smaller threads of the GPU exploit it better. Due to larger variations, the deep learning scheduler does not calculate optimal M choices, hence the scheduler exhibits some overhead over the GPU for some cases. Overall, the framework is 31% better than a GPU-only and 75% better than a Xeon-Phi-only setup.

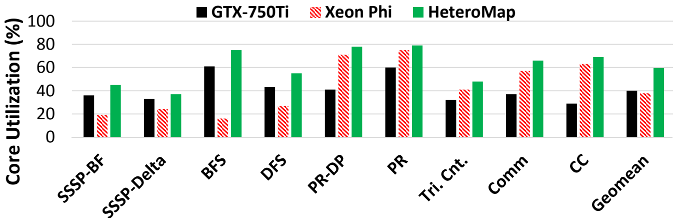


Fig. 13: Core Utilization benefits averaged for various inputs for each benchmark.

HeteroMap vs. Manual Tuning: HeteroMap is within 10% performance of an ideal case utilizing manual tuning, which shows significant accuracy improvement compared to prior learning works. The overhead of the HeteroMap framework rises in some cases where performance is competitive. Examples such as PR-LJ perform equally well in both accelerators, which causes HeteroMap to select slightly different intra-accelerator choices, leading to some overheads.

C. *Understanding Energy & Utilization Variations*

HeteroMap is also trained for the energy objective. Figure 12 shows normalized energy (normalized to the maximum energy for B,I combinations) for various benchmarks. Geomeans of energy are taken across the different inputs for each benchmark. The Xeon Phi has a larger power rating compared to the two GPUs, and hence it dissipates more energy. Certain inputs take more time to complete on the GPU, which adds to its energy woes. HeteroMap reduces energy usage in this case from (0.15, 0.16) to just 0.06, by a factor of 2.4×. This is fairly close to the ideal case (0.03). This favors the deployment of HeteroMap in energy constrained environments.

HeteroMap also improves core utilization by selecting optimal architectural choices, and this is the main reason performance benefits are exhibited. Figure 13 shows the raw core utilization (%), averaged across cores and inputs, for each benchmark. Utilization for throughput-dependent benchmarks such as SSSP is low for the Xeon Phi as its cores spend most

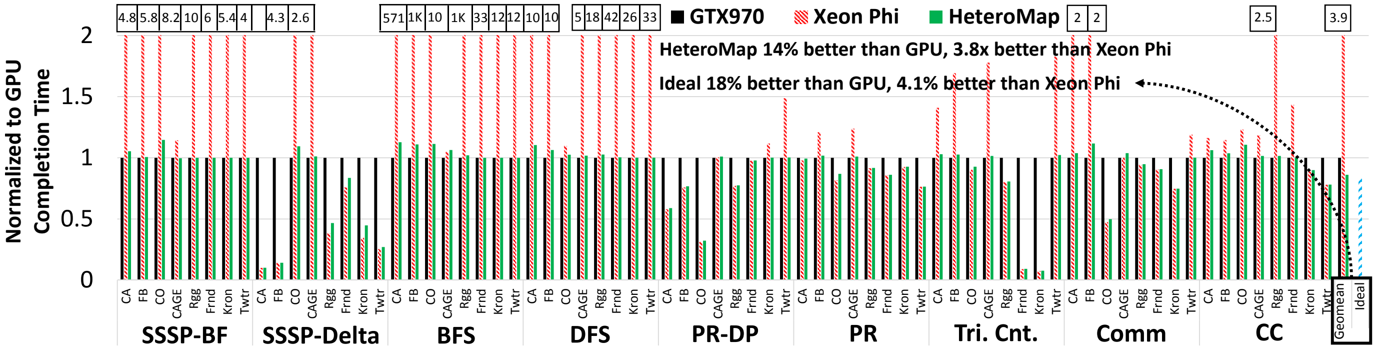


Fig. 14: Scheduler Comparisons for various Graph Workloads-Input Combinations (All results normalized to the GTX970Ti GPU implementation) (Higher is worse). Note that Optimal Choices change when compared to the GTX750Ti in Figure 11.

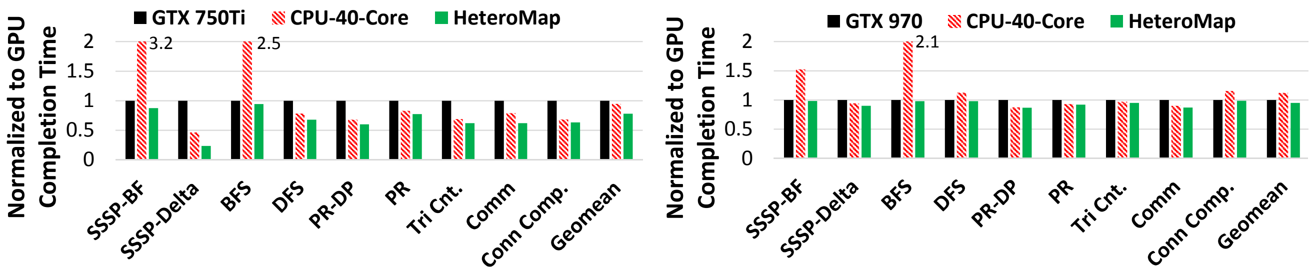


Fig. 15: Geomean results averaged for different inputs for each benchmark for the 40-core CPU. All results are normalized to the GPU implementation.

of their time waiting for low-locality memory accesses. GPUs can hide such latencies via thread switching, and thus have better core utilization. HeteroMap improves the geomean by 20% over both machines, which is primarily due to optimal accelerator and intra-accelerator threading selections.

D. Changing Fixed Accelerator & Memory Sizes

Accelerator Changes: As accelerators change across various HPC setups, a stronger GTX-970 GPU having more resources is considered, increasing concurrency choices. A weaker GPU was compared first to show whether the GPU architecture inherently benefits benchmark-input combinations or not (which was shown to be the case). Machine learning models are re-learned for this architectural change. As shown in Figure 14, benchmark trends compared to the smaller GPU remain mostly the same, with concurrent workloads such as SSSP-BF still performing well on the GPU. Comparing other workloads that were only slightly better on the Xeon Phi before, such as TRI-LJ, the stronger GPU performs better. Overall, HeteroMap outperforms a GPU-only case by 14% and a Xeon-Phi-only case by 3.8×, as the magnitude by which the GPU outperforms Xeon Phi in some cases is higher compared to the GTX-750. But the Xeon Phi still beats the GTX-970 for other combinations, and 14% is remarkable as the GTX-970 has twice the single-precision compute power.

A 40-core multicore CPU is also compared with the GTX-750Ti and the GTX-970 GPUs. Figure 15 shows the normalized to GPU completion times averaged for all inputs for a particular benchmark. The GPUs in both cases are seen to outperform the CPU for highly parallel benchmarks such as SSSP-BF and BFS. For other benchmarks, the CPU outperforms the weaker GTX-750 GPU. In the case of the GTX-970, the GPU performs better than the CPU for DFS and Conn. Comp. This is because the stronger GPU has larger caches and more cores than the smaller GPU, allowing the two benchmark's indirect accesses to be able to perform better in the GTX-970. The 40-core multicore outperforms the GTX750 by 3% for a 2 GB memory size for each accelerator. For the case with the GTX-970, the GPU outperforms the 40-core multicore by 10% for a 4 GB memory size for each accelerator. Using HeteroMap, performance gains of 22% and 5% are acquired over the GTX-750 and the GTX-970 respectively. HeteroMap achieves these gains as it selects the optimal accelerator for each benchmark-input combination. Averaging across inputs, HeteroMap picks the better result of the two accelerators to produce better results than either of the two accelerators for each benchmark.

GPU-Xeon Phi memory size sensitivity: Main memory is an important parameter that one can re-architect to change a system. However, in our system we only sweep memory sizes that the accelerators support i.e., up to 2-4 GB for GPUs, and up to 16GB for the Xeon Phi. Figure 16 shows various memory sizes for the target accelerators. Error bars show variation in performance of either accelerator. Geometric mean of all the benchmark-input combinations is taken for a particular memory size (GPU, Xeon Phi). The y-axis shows the completion time normalized to the max (the upper error bar for (1,1)), with the geomean of the average of all combinations normalized to the GPU. The overall trend is that the Xeon Phi performs better when it is exposed to its full main memory

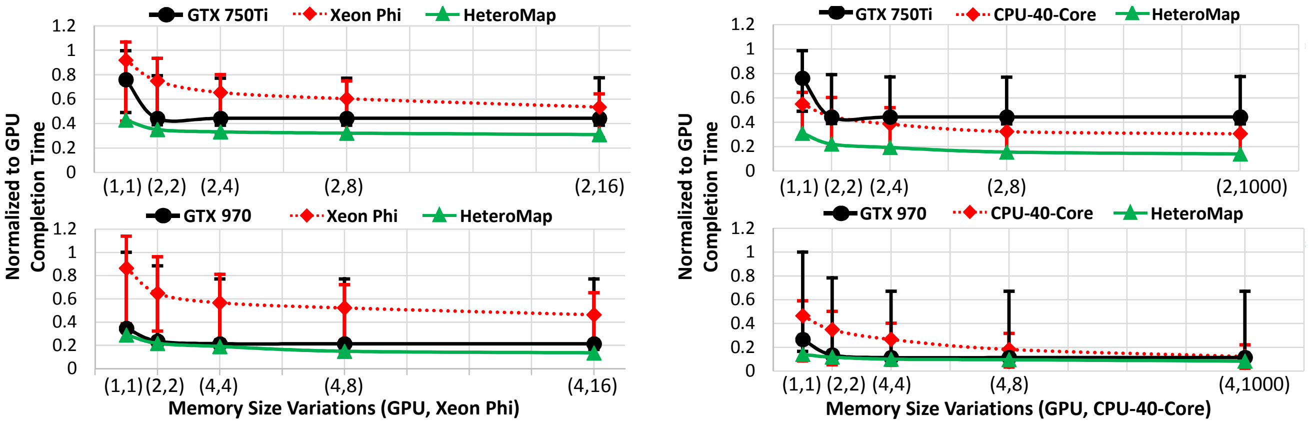


Fig. 16: Geomean Memory Size Variations for various target accelerators

compared to both GPUs (30% better than the GTX-750TI and 15% better than the GTX-970). This is because it can exploit the full memory bandwidth and size, forgoing the need for memory transfers compared to GPUs. Even though GPU memory saturates at 2 or 4 GB, keeping the GPU performance constant after maximizing memory, the Xeon Phi's performance is still off by 10-20%. HeteroMap is able to exploit this memory variation as an addition to the vector $\vec{M}$, and is able to learn with performance benefits higher than acquired with limited main memory for the Xeon Phi.

GPU-40-Core CPU memory size sensitivity: This work also compares a 40-core multicore CPU in conjunction with GPU accelerators. Figure 16 shows performance with various memory sizes for this setting. The 40-core CPU performs better than the two GPUs on average. The CPU also improves when it is exposed to its full memory capacity, which allows larger graphs such as Twitter and Friendster to fit in its main memory. The CPU improves over the GTX-750Ti by 18%, and over the GTX-970 by 5%, for the maximum memory sizes. Although HeteroMap improves slightly in the geomean case over the GPU, there are many individual cases where it improves over both machines by up to 3×. The primary reason why the 40-core CPU is better than the GTX-750 and the GTX-970 is that the CPU runs at a higher frequency (2.3 GHz vs. GTX750's 1.3 GHz and GTX-970's 1.7 GHz). Other reasons that improve the CPUs performance include its better caching capabilities and stronger core pipelines.

VIII. Related Work

Prior works in performance prediction mainly involve operating system runtimes [38] [40] [41] to improve utilization in single machine setups. Such works do not analyze graphs and *input dependence* due to space complexity. There is a plethora of work that optimizes unary single-accelerator CPU-GPU systems [42] [6]. HeteroMap differs from these works to justify how architectural aspects across accelerators can be exploited in real-time to overcome unary setup limitations. Schemes proposed in this paper can be deployed on top of runtimes (OpenMP utilized in this paper). Some works generate predictive models [31] to optimize for inputs [43], and optimize intra-machine choices [26]. However none of these works generate analytical models or optimize for different *competitive* accelerators, such as GPUs and Xeon Phis, for graph analytics. Such multicores have many more concurrency choices compared to CPUs due to more thread count, placement, dynamic scheduling, and synchronization, combinations [44]. Prior predictive models also suffer from high error rates (e.g. 26.9% in [31]), making QoS [45] an issue. Therefore a proper learning analysis is necessary to enable real-world deployment aspects.

Other works in auto-tuning such as PetaBricks [46, 9] and OpenTuner [10] exploit algorithmic choices, and have not explored architectural variations. Moreover, as algorithmic spaces constitute higher complexities, learning takes unreasonable amounts of time [47]. Regression based autotuners [31] have lower complexities, but these are still high enough to defer near real-time deployment. Thus developing runtimes for optimizing such spaces in graph processing remains an intractable problem for now, and the optimal way is to learn intelligently on a limited number of choices to configure accordingly. Such works also lack characterization of graph workloads as targeted in this paper, which are more unpredictable due to input dependencies. However, OpenTuner is used for off-line training in this work, as it is used to exhaustively search the complex B, I, M choice space.

IX. Conclusion

This paper presents a prediction framework, **HeteroMap**, for a multi-accelerator architecture that optimizes architectural choices for real-time processing of graph analytics. When inter- and intra-accelerator and graph benchmark-input choices are coupled together, the near-optimal choice selection problem is very complex. This work not only quantifies graph benchmark and input choices, but also relates them to machine choices in a multi-accelerator system using an analytical model and automated machine learning predictors. Automation of the framework is done using off-line training and on-line evaluation to select an optimal accelerator and its architectural choices. Evaluations show performance gains of 5% to 3.8× when comparing single accelerators, and the proposed learner is within 10% of an ideal case, which is a boost in predictive concurrency analysis compared to prior works.

Acknowledgments

This work was funded by the U.S. Government under a grant by the Naval Research Laboratory. This work was also supported by the National Science Foundation (NSF) under Grant No. CNS-1718481, and in part by Semiconductor Research Corporation (SRC). We would like to thank Christopher Hughes (Intel Corporation) and Jose A. Joao (arm Research) for their constructive feedback and support.

References

[1] M. Ahmad, K. Lakhsminarasimhan, and O. Khan, "Efficient parallelization of path planning workload on single-chip shared-memory multicores," in *Proceedings of the IEEE High Performance Extreme Computing Conferenc*, ser. HPEC '15. IEEE, 2015.

[2] N. Satish and et. al., "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.

[3] Cray-Inc., "http://www.cray.com/blog/characterization-of-an-application-for-hybrid-multimany-core-systems/," 2015.

[4] 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 International Symposium on*, May 2015, pp. 1092–1098.

[5] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IEEE Int. Symposium on Workload Characterization (IISWC)*, Sept 2013, pp. 185–195.

[6] S. Fang and et. al., "Performance portability across heterogeneous socs using a generalized library-based approach," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, Jun. 2014.

[7] 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.

[8] 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.

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

[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] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Int. Symposium on Workload Characterization*, ser. IISWC, 2015.

[12] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single-source shortest paths with the parallel boost graph library," *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, pp. 219–248, 2006.

[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[14] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 12–25.

[15] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, "Graphit - A high-performance DSL for graph analytics," *CoRR*, vol. abs/1805.00923, 2018.

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

[17] 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.

[18] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, Sept 2012, pp. 1–5.

[19] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on gpus," vol. 11, no. 1. VLDB Endowment, Sep. 2017.

[20] A. B. Adcock, B. D. Sullivan, O. R. Hernandez, and M. W. Mahoney, "Evaluating openmp tasking at scale for the computation of graph hyperbolicity," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 71–83.

[21] T. Suzumura and K. Ueno, "Scalegraph: A high-performance library for billion-scale graph analytics," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 76–84.

[22] J. Leskovec and R. Sosivc, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

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

[25] 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.

[26] 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.

[27] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic galois: On-demand, portable and parameterless," in *The 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS. New York, NY, USA: ACM, 2014.

[28] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[29] G.-B. Huang, D. H. Wang, and Y. Lan, "Extreme learning machines: a survey," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, Jun 2011.

[30] S. Tamura and M. Tateishi, "Capabilities of a four-layered feedforward neural network: four layers versus three," *IEEE Transactions on Neural Networks*, vol. 8, no. 2, pp. 251–255, Mar 1997.

[31] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 725–737.

[32] 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.

[33] 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.

[34] "Measuring power on intel xeon phi product family devices, https://software.intel.com/en-us/articles/measuring-power-on-intel-xeon-phi-product-family-devices," 2015.

[35] "Ubuntu-manuals : powerstat - a tool to measure power consumption," 2015.

[36] Nvidia, "https://docs.nvidia.com/cuda/profiler-users-guide/index.htmls," in *CUDA*, 2018.

[37] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[38] 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.

[39] G.-B. Huang and H. A. Babri, "Upper bounds on the number of hidden neurons in feedforward networks with arbitrary bounded nonlinear activation functions," *IEEE Transactions on Neural Networks*, vol. 9, no. 1, pp. 224–229, Jan 1998.

[40] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 169–180.

[41] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 376–387.

[42] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *The 2011 Int. Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 78–88.

[43] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the Int. Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 149–160.

[44] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan, "Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 254–264.

[45] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 97–110.

[46] J. Ansel and et. al., "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.

[47] D. Tarjan, K. Skadron, and P. Micikevicius, "The art of performance tuning for cuda and manycore architectures," *Birds-of-a-feather session at Supercomputing (SC)*, 2009.