GRapid: a Compilation and Runtime Framework for Rapid Prototyping of Graph Applications on Many-core Processors

Da Li[1], Srimat Chakradhar[2], Michela Becchi[1]
[1]University of Missouri-Columbia, [2]NEC Laboratories America
da.li@mail.missouri.edu, chak@nec-labs.com, becchim@missouri.edu

***Abstract*—Many applications use graphs to represent and analyze data, but the effective deployment of graph algorithms on many-core processors is still a challenge. Many-core devices offer higher peak performance than multi-core devices; however, many-core programming is still a specialized skill. While compilation and runtime frameworks for parallelizing graph applications on multi-core CPUs exist, there is still a need for comparable frameworks for many-core devices.**

We propose GRapid: a compilation and runtime framework that generates efficient parallel implementations of generic graph applications for multi-core CPUs, NVIDIA GPUs and Intel Xeon Phi. Applications are expressed using a platform-agnostic programming API. Our source-to-source compiler performs platform-specific code transformations and optimizations, handles data transfers between the host and the coprocessor, and generates several functionally-equivalent code variants for the target platform. Our runtime library provides an efficient dynamic memory management scheme for applications where the graph topology is modified during processing. We used the proposed framework to rapidly generate efficient implementations of four graph applications on different target processors. Such rapid prototyping and re-targeting capability has obvious programmability advantages, but it can also be used to quickly identify target processors that better match the application's computational needs.

Keywords—Graph applications, many-core processors, source-to-source transformation, runtime library

I. INTRODUCTION

GPUs have been widely used to accelerate a variety of applications [1] for over a decade. Intel recently released a many-core processor (Xeon Phi), which has more than fifty x86 cores, each supporting four hardware threads. The peak double-precision performance of high-end many-core devices from NVIDIA, AMD and Intel is well above 1 Teraflops.

Although many-core processors are widely used, they are still relatively difficult to program, since they require programmers to be familiar both with parallel programming and with the features and the operation of these hardware platforms. This complexity is aggravated by the variety of software stacks used by the various many-core platforms.

Parallelization of regular applications (such as those operating on dense matrices and vectors) on many-core processors has been extensively investigated. However, parallelization of irregular applications continues to be a challenge. Irregular applications are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence, and a degree of parallelism that is known only at runtime. Many established and emerging applications are irregular because they are based on irregular data structures, such as graphs. Even in the data analytics domain, graph databases (like ArangoDB and Oracle Spatial and Graph) are emerging as alternative to relational databases.

In this work, we intend to fill this gap and propose a compilation and runtime framework (Figure 1) for the effective deployment of *generic* graph applications on many-core processors (GPUs and the Intel Xeon Phi). Note that our framework also produces multi-threaded code for multi-core CPUs. Quite unlike previous work [2, 3], we consider graph applications that use *static* or *dynamic* datasets, and we free the programmer from the need to write specific parallel kernels for GPUs and the Intel Xeon Phi. Our framework hides the complexity and heterogeneity of the underlying hardware and software stack from the programmer. The programming API exposed to the user is *platform-agnostic,* and includes a set of platform-independent sequential and parallel constructs. Our source-to-source compiler converts the graph and the containers (sets, multi-sets and queues) into internal, platform-specific data structures for multi-core CPUs, Intel Xeon Phi and NVIDIA GPUs. It then uses iterator-based templates to parallelize graph processing. The compiler generates different functionally-equivalent *code variants* for the target platforms. These variants differ in the parallelization strategy and in the optimized data structures on which they rely. Our runtime system, designed to support automatic selection of code variants and parameter tuning (based on profiling), also includes support for efficient dynamic memory management.

Unlike previous work [4-11], we do not aim to optimize a specific algorithm on a particular many-core processor, but to automate the development of high-performance graph applications on many-core platforms. By leveraging our platform-agnostic programming API, the application developer delegates the complex task of tailoring the application to a particular platform to our toolchain.

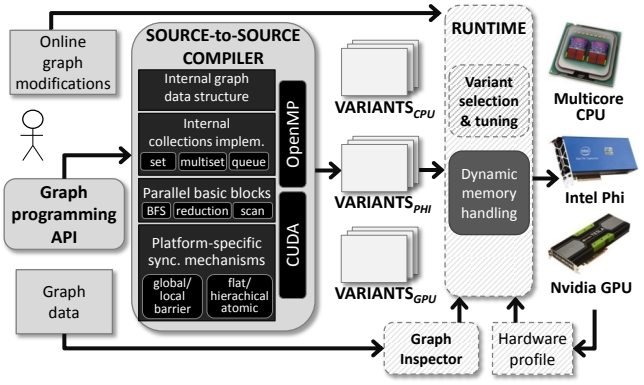

Fig.1. Proposed graph processing system. Dashed blocks represent components not detailed in this paper.

We make the following contributions:

- We design source-to-source transformation techniques to automatically generate parallel code for different many-core platforms (GPUs and the Intel Xeon Phi) starting from a platform-agnostic graph programming API.
- We include programming and runtime support for dynamic memory allocation, and study the effect of synchronization on the performance of our runtime library.
- We evaluate our toolchain on four applications: BFS, A-DFA compression, PageRank and DFA generation. These workloads have different computational patterns and complexity. We report lessons learned from the parallelization of these.

II. GENERAL DESIGN & METHODOLOGY

Graph algorithms can be represented as a sequence of iterative steps. At each step, the algorithm performs some work on the elements of a *working set* and updates the working set (typically, by visiting neighbors of an element) by adding or removing elements. In every iteration, the elements of the working set may be processed in parallel (although synchronization mechanisms may be required to control concurrent data accesses). This computational pattern maps naturally to the *bulk synchronous* [12] style of parallelism.

Figure 1 describes our proposed graph processing system. The *programming interface* consists of a high-level graph API and a set of platform-agnostic, sequential and parallel constructs allowing the user to define generic graph applications. The *source-to-source compiler* generates different code variants for multi-core CPUs, Intel Phi coprocessors and NVIDIA GPUs. These code variants may differ in several aspects: from the type of parallelization performed, to the implementation of the underlying data structures [13], to the handling of nested parallelism, and more. The generated code is written in OpenMP and CUDA, and it uses the offload execution model on the Intel Phi. During code generation, the graph and the containers (sets, multi-sets and queues) are converted into internal, platform-specific data structures. In addition, existing parallel basic blocks are used for common primitives such as *reduction*, *sort* and *scan*. Parallelization is enabled by the presence of parallel iterators, which must be explicitly inserted in the code by the programmer. The compiler automatically handles synchronizations associated with the graph, the iterators and the containers. Synchronizations associated with custom data structures must be explicitly indicated by the programmer using high-level, platform-independent synchronization primitives, which are converted into platform-specific synchronization mechanisms. Finally, the *runtime system* supports two important functions: (i) selecting the most suitable code variant depending on the characteristics of the application, the dataset and the underlying platform, and (ii) supporting dynamic memory allocation. In this paper, we focus on the second function and propose a runtime library for dynamic memory management. In addition, in Section VII we provide guidelines to efficiently match the application to the hardware platform.

```
GRAPH API
graph/node/edge
Default attributes
graph: nodes, edges, root, num_nodes, num_edges, directed
node: (in_/out_)neighbors, (in_/out_)edges, (in/out)degree,
      level
edge: left, right, weight; primitive: node mate(node)
Methods to define/manipulate application-specific attributes
void addAttr(graph/node/edge, attr_name, type, default_value);
void setAttr(attr_name, value);
value getAttr(attr_name);

CONTAINER DATA STRUCTURES
set: void add(item), void remove(item), bool include(item),
     bool empty(), int size(), void clear(), bool equal(set)
oset: primitives of set; item first(), item next(item)
multiset, omultiset: primitives of set/oset, int occurrences(item)
queue: void push(item), item pop(), item front(), int size(),
       bool empty(), item next(item), void clear()

ITERATORS
sequential:
    while(condition [; dynamic_update(set)])
    for(datatype item:domain [; dynamic_update(set)])(filter)
parallel:
    foreach(datatype item:domain [; clear domain])(filter)
    inBFS(var: domain from source_node)

DYNAMIC MEMORY MANAGEMENT PRIMITIVES
newGraph
addNode/deleteNode
addEdge/addDirectEdge/deleteEdge
new/delete

PARALLEL PRIMITIVES
item reduction(container, operator)
void scan(in_container, out_container, operator)
void sort(in_container, out_container)

SYNCHRONIZATION PRIMITIVES
barrier
critical{}

RUNTIME PRIMITIVES
void commit(bool) - commits a set of changes to the graph and,
                    if parameter is true, to the working set
void rebalance() - rebalance an extended CSR representation
```

Fig. 2. Graph programming API.

III. PROGRAMMING API

We aim to provide the programmer with a familiar and easy-to-use programming interface, similar to existing ones designed for CPUs. To this end, we extend Green-Marl's API [14] with dynamic memory allocation and runtime primitives to support applications that use dynamic data sets (Green-Marl assumes static data sets), and we borrow some containers (ordered and unordered sets) from the Galois's system [15].

The resulting programming interface is summarized in Figure 2.

Graph API: The graph API includes the abstract data structure and high-level primitives that can be used by the programmer to define and manipulate graphs. Graphs, nodes and edges have *default* attributes which are part of the API. For example, each graph consists of a set of *nodes* and *edges*, can be directed, and may have a root node. Each node has a set of neighbors and of (outgoing and incoming) edges, and a *level*. Each edge has a *left* and *right* vertex and potentially a *weight*. These basics data structures can be extended via user-defined, *application-specific* attributes. Such attributes can be defined using the *addAttr* primitive, and their value can be set and queried using the *setAttr* and *getAttr* methods, respectively.

Container data structures: A variety of containers (*unordered and ordered sets* and *multisets*, and *queues*) can be used to build graph algorithms (for example, to represent the working set). These containers come with a set of access and manipulation primitives (*add*, *remove*, *empty*, etc.) and can apply to generic objects. Internally, containers operate on numeric data types and pointers to objects and are mapped to platform-specific, thread-safe data structures.

Iterators: Iterators provide the ability to define loops. They

can be of two kinds: sequential (*for* and *while*) and parallel (*foreach* and *inBFS*). Parallel iterators allow the programmer to expose parallelism within the application. The *inBFS* iterator (from Green-Marl [14]) modifies the *level* attribute of the nodes.

Dynamic memory management primitives: Dynamic memory allocation primitives can be graph-specific (e.g., *newGraph*, *add*/*deleteNode*, *add*/*deleteEdge*) or general-purpose (e.g., *new* and *delete*). The former map to the internal, optimized graph representation; the latter can be used for containers or application-specific, user-defined data structures. All these primitives are internally mapped to a *custom_malloc* function and handled within our runtime system.

Parallel primitives: Commonly used parallel primitives on containers (*reduction*, *scan* and *sort*) are internally mapped to platform-specific, optimized, thread-safe implementations.

Synchronization primitives: The source-to-source compiler automatically handles synchronization related to the graph data structure, containers and iterators. However, application-specific, user-defined data structures may also require synchronized access. This kind of synchronization must be explicitly indicated by the programmer through high-level primitives (*barrier* and *critical*). These primitives are internally mapped to platform-specific synchronization mechanisms.

Runtime primitives: Runtime primitives can be invoked when the graph data structure is modified by external intervention. Specifically, *commit* is used to commit a set of graph modifications to the runtime system; after a *commit*, the required modifications are applied to the internal graph and possibly incorporated in the working set. The *rebalance* primitive allows re-optimizing the internal layout of the graph data structure (see Section V for more detail).

IV. CASE STUDIES

We consider two kinds of applications: graph analytics and general purpose graph processing. Specifically, we target three categories of workloads.

Graph analytics on static datasets: These *read-only* algorithms perform analysis tasks on graphs that do not change over time or that change so infrequently to justify rerunning the algorithm on the whole graph when this happens.

Graph analytics on dynamic datasets: These *read-only* algorithms perform analyses on graphs that may change over time. The graph itself is not modified by the algorithm, but by external events. For instance, in social network, graphs constantly change due to "friend" and "unfriend" acticities.

General purpose graph processing: These *read-write* algorithms perform various types of general purpose computations that may modify the structure of the underlying graph. For example, subset construction [16], which transforms a non-deterministic finite automaton (NFA) into a deterministic one (DFA), is a general purpose read-write algorithm.

```
1. Procedure A-DFA (graph dfa, int d){
2.   inBFS(n: dfa.nodes from dfa.root){
3.     int max = 1;
4.     n.setAttr('default', UNDEFINED);
5.     foreach(node p:dfa.nodes)
6.       if (p.level < n.level & n.level-p.level ≤ d){
7.         int cn = common_neighbors(n,p);
8.         if (cn > max){
9.           max = cn;
10.          n.setAttr('default', p);
11.} } } } }
```

Fig. 3. A-DFA compression algorithm.

We briefly describe two algorithms used as example workloads and illustrate our programming API on them.

A-DFA: A-DFA [17] is a compression algorithm used to reduce the memory footprint of DFA accepting large sets of regular expressions. For simplicity, in this paper we focus on the computation of the application-specific *default* transition attribute. Thus, the A-DFA algorithm shown in Figure 3 is *read-only* and operates on a static graph. Specifically, the algorithm visits a DFA graph in BFS manner (line 2). It compares each node with every other node at lower depth (skipping *d* levels) and selects the node with more transition commonality as *default* target state. Compared to BFS, A-DFA presents a non-trivial computation phase. In fact, the work executed at every step of the BFS traversal (lines 3-11) includes control-flow operations and scattered memory accesses to the whole DFA graph. We also notice that this algorithm exhibits a two-level parallelism (*inBFS* at line 2 and *foreach* at line 5).

PageRank: PageRank (Figure 4) is commonly used in search engines to rank webpages based on their significance. The *rank* attributes are initialized to a default value upon creation. In each iteration of the main loop (line 2), the ranks are updated according to the *outdegree* of connected nodes. Specifically, every node distributes its rank evenly to its outgoing neighbors (lines 5-6). PageRank terminates after all ranks converge (i.e., their variation falls below a given threshold *delta*). PageRank is a *read-only* algorithm that can operate on either static or dynamic graphs since webpages can be created and modified by user intervention. The user notifies the runtime system by issuing a sequence of *addNode* and *addEdge* (and corresponding *delete* operations) followed by a *commit*. The *commit* causes the involved nodes to be added to the working set *ws*. External modifications to *ws* are processed in the next iteration. The *dynamic_update* primitive in the iterator at line 2 indicates that external updates should be incorporated in *ws* at the beginning of each iteration. This can happen while PageRank is running or when it is terminated (in which case it will be reactivated). We note that the ranks are usually double precision floating point numbers, thus requiring the use of floating point arithmetic (on GPUs, floating point is slower than integer arithmetic).

DFA construction: DFA are typically used by applications performing regular expression matching. In this context, regular expressions are initially compiled into a NFA. Then, the NFA is transformed into DFA through subset construction [16] (Figure 5). Like A-DFA, DFA construction proceeds in BFS manner and exhibits a two-level parallelism (*inBFS* at line

```
1. Procedure PageRank(graph g, double delta){
2.   for(set ws=g.nodes; !ws.empty(); dynamic_update(ws)){
3.     foreach(node n: ws){
4.       n.setAttr('nr',0);
5.       for(node m: n.in_neighbors)
6.         n.incAttr('nr', m.getAttr('rank')/m.outdegree);
7.     }
8.     foreach(node n: ws; clear ws){
9.       if (abs(n.getAttr('nr')-n.getAttr(rank))>delta)
10.        ws.add(n.out_neighbors);
11.      n.setAttr('rank',n.getAttr('nr'));
12.} } }
```

Fig. 4. PageRank algorithm.

```
1.  Procedure DFA_construction (graph nfa, graph dfa){
2.    dfa = newGraph();
3.    subset mapping = new subset();
4.    dfa.root = dfa.addNode(mapping.update(new set(nfa.root)));
5.    inBFS(n: dfa.nodes from dfa.root){
6.      set *nfa_subset = mapping.reverse_lookup(n);
7.      foreach(char c: alphabet){
8.        set *target = new set();
9.        for(node m: nfa_subset)
10.         for(edge e: m.out_edges)
11.           if (e.getAttr('symbol') = c) target.add(e.mate(m));
12.       if (!mapping.lookup(target)){
13.         critical{
14.           node state = dfa.addNode(mapping.update(target));
15.       } }
16.       else
17.         delete target;
18.       edge ne =dfa.addDirectEdge(n,state);
19.       ne.setAttr('symbol',c);
20. } } }
```

Fig. 5. DFA (subset) construction algorithm.

5 and *foreach* at line 7). Again, the work performed in each iteration (body of *inBFS* loop at lines 6-20) contains control-flow operations and irregular memory accesses. However, DFA construction involves an additional complexity: it modifies the DFA graph (in fact, it creates it). As can be seen, DFA construction involves dynamic memory operations on the DFA graph (lines 2, 4, 14 and 18), on sets (lines 4, 8 and 17) and on the custom *subset* data structure (lines 3, 4 and 14). The latter has a complexity hidden within its *lookup* and *update* primitives. Briefly, *subset* is a double linked-list data structure used to verify in linear time if a subset belongs to a power set. The programmer can make the code parallelizable by using a *critical* section (lines 13-15): this is an example of *coarse-grained synchronization*. In alternative, the programmer can provide a thread-safe implementation of the *subset* data structure optimized for different platforms (thus allowing custom *fine-grained synchronization*).

V. COMPILER DESIGN

Once graph algorithms are expressed by using our platform-agnostic API, our source-to-source compiler generates different code variants for GPU, Intel Phi, and multi-core CPU. We describe the main aspects of the compilation process.

A. Data Structure Design

We encode graphs by using an extension of the *compressed sparse row* form (CSR), a commonly used format for sparse data structures [4]. To support dynamic graphs, the basic CSR data structure is extended into a hierarchical-CSR (Figure 6). Initially, we overprovision both the level-1 *nodes* and *edges* arrays. In the *edges* array, we pre-allocate a default number of edges for each node (blank slots in level-1 *edges* array in Figure 6). The optimal provision size depends on the characteristics of the graph. Reserving larger space can improve the insertion performance at the cost of wasting memory. At each deletion, we invalidate elements. We use the free elements of the *nodes* and *edges* arrays for insertions. When the level-1 *nodes* array overflows, we allocate a level-2 *nodes* array. Similarly, when the portion of the *edges* array allocated to a node *n* overflows, we start inserting the new edges of *n* in a level-2 *edges* array. We repeat this operation recursively: if the level-2 edges/nodes array overflows, a level-3 one is allocated. For each *node*, one extra variable is needed to store the "pointer" (x, y, p, q in Figure 6) to its allocated *edges* array. For each allocated array, the last element of level-*x* block (the shadow area in Figure 6) stores a "pointer" to the level-*(x+1)* block (null-pointer if such block has not been allocated). Also, for fast insertion into any level-*x* block, the first element (solid black area in Figure 6) records the used space in the block. The size of each block can be fixed (e.g. 128B, which is cache and memory friendly) or increase quadratically. The resulting hierarchical-CSR allows efficient dynamic insertions and deletions but, due to its nested structure, is less efficient to traverse. The runtime can periodically call the *rebalance* primitive to transform the hierarchical-CSR into pure CSR form. Since all the allocations are using the memory pool managed by our runtime, instead of residing in the OS heap, it increases the locality. More importantly, in our design, all the pointers to the allocated arrays actually store the indices in the memory pool, not the virtual address. This design enables seamless computing across platforms: moving the hierarchical-CSR between platforms by simply copying the basic CSR and the content in the memory pool. For container data structures, we use basic blocks proposed and discussed in previous work [4, 6, 8].

B. Code Generation

We now describe our code generation process along with the platform-specific transformations we perform.

General Design – Users encode functions corresponding to the "hot spots" in their applications using our programming API. Our source-to-source compiler then transforms these user-defined functions into C++ wrapper functions containing platform-specific code. In the case of multi-core CPUs, code regions associated with parallel iterators are translated into parallel regions through the insertion of OpenMP directives. GPUs and Intel Phi coprocessors require handling also the data transfers between host and device. In particular, the compiler generates data transfers for graphs and user-defined data structures declared outside the parallel regions and referenced inside them. In the case of the Intel Phi, parallel regions are handled using OpenMP directives and placed inside offload regions surrounded by the "*#pragma offload*" directive. The primitives to allocate variables on the coprocessor and move data between host and device are inserted along with the offload directives. In the case of GPUs, regions of code associated to parallel iterators are translated into parallel kernels. In this process, elements of the working set are

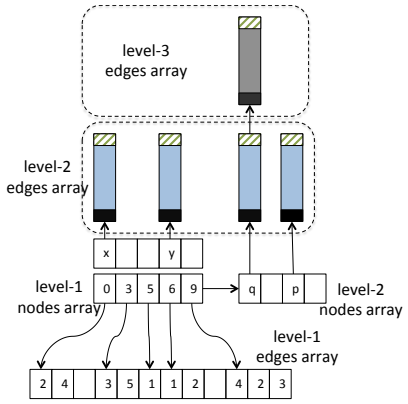

Fig. 6. Hierarchical-CSR with level-2, level-3 nodes/edges arrays.

mapped onto threads or thread-blocks (i.e., thread- and block-based mapping [13]) producing different code variants. The compiler then generates code for data transfers (using the *cudaMalloc* and *cudaMemcpy* primitives) and kernel launches.

Handling of Nested Parallel Iterators – The presence of nested parallel iterators enables different code variants and can be handled differently on various accelerators.

The Intel Phi has a flat hardware parallelism. However, the two-level nesting of A-DFA and DFA construction leads to different alternatives on the placement of the offload and OpenMP parallel directives. Specifically, we can: (i) place a synchronous offload at the level of the sequential *for* and the parallel directive at the level of the outer *inBFS* iterator; (ii) place both the synchronous offload and the parallel directive at the level of the outer inBFS iterator; (iii) use an asynchronous offload and a parallel directive on one of the parallel iterators (thus launching several parallel offloads concurrently). We experimentally found that the offload overhead makes the first code variant preferable.

NVIDIA GPUs present a two-level hardware parallelism. Therefore, two-level nested iterators (as in Figure 3 and 5) can be naturally handled by using block- and thread-based mapping for the outer and inner parallel iterators, respectively. On the other hand, using thread-based mapping on the outer-loop will cause serialization of the inner-loop. In the presence of three nested parallel iterators, an additional level of parallelism can be achieved by using multiple streams and the parallel kernel execution feature available starting from the Fermi architecture [18]. Given its massive hardware parallelism and its Hyper-Q technology, code variants using concurrent streams are particularly suited to Kepler GPUs. Kepler GPUs also allow an additional level of nesting through dynamic parallelism. However, we experimentally found that the overhead for launching nested kernels is significant, making it difficult to achieve good performance by using this feature.

Other Accelerator-specific Optimizations – To reduce the communication overhead, we leverage the compiler analysis techniques proposed by [19] to identify data reused by subsequent parallel kernels with no intermediate CPU read or write access. These data can be stored persistently on the coprocessor, thus avoiding unnecessary data transfers between host and device. On the Intel Phi, we use the *alloc_if* and *free_if* clauses to control the allocation of data and the *in*, *out*, and *nocopy* modifiers to avoid unnecessary data transfers.

On the Intel Phi, the use of vectorization can greatly help performance. Vectorizable code can come from either *foreach* or *for* iterators. In the PageRank algorithm (Figure 4), for example, the *for* loop at line 6 is a good candidate for vectorization. In the case of *for* loops, however, data dependences must be resolved (since these iterators are sequential). To this end, we rely on a feature of the Intel compiler: inserting a *#pragma ivdep* before a loop allows the Intel compiler to resolve conservatively assumed data dependences and possibly generate vectorized code.

VI. RUNTIME LIBRARY DESIGN

The runtime system has essentially two functions (Figure 1): dynamically selecting and tuning the code variant that better fits the characteristics of the target dataset and the hardware profile, and handling dynamic memory allocation. Variant selection can be done based on profiling information and by monitoring the size of the working set, and dynamically selecting the code variant that better fits that level of parallelism [13]. Due to space constraint, in this paper we only describe dynamic memory management

A. Dynamic Memory Management

NVIDIA GPUs lack of operating system support and of an efficient mechanism to handle dynamic memory allocation within parallel kernels. Starting from the Fermi Architecture NVIDIA has added support for the *malloc* call. However, the use of *malloc* on GPU has two limitations: first, it leads to inefficient code for small size allocation; and second, it fails in the presence of large numbers of *malloc*. We observed that, when using system *malloc*, DFA construction fails even on small graphs (about 30k nodes). To circumvent this problem, we have introduced a custom memory management scheme for GPU. We have ported this mechanism to multicore CPU and Intel Phi, and have compared our proposed scheme with the direct use of the system *malloc*.

General design: Our basic idea is to use a memory pool with fixed-size blocks. Specifically, we start by pre-allocating a single block with a handle pointing to the next free position within the block. Each *custom_malloc* call will obtain the requested number of bytes from the block, and will cause the handle to be incremented accordingly. When the current block fills up, a new block is allocated by the runtime.

Our proposed solution use multiple locks (one per thread-group) to reduce lock contention. We use two types of blocks (with separate handles): *permanent* and *temporary* blocks. Permanent blocks have the application lifetime, whereas temporary ones have the lifetime of an iteration of the algorithm. Variable de-allocations within the temporary blocks are deferred until the end of the corresponding iteration, when temporary blocks are cleared by resetting their handle. Compile-time code analysis determines which kind of *custom_malloc* to perform for each dynamically allocated variable and the runtime moves temporay data to permanent blocks when needed. This significantly reduces fragmentation and de-allocation cost, which distinguishes our dynamic memory management from other proposals. We compared our scheme with halloc (GPU allocator) in Section VII.

Intel Phi/CPU implementation: Since the Intel Phi and the CPU have a flat thread organization, thread grouping is done based on the thread identifiers. Our experimental results show that on CPU the system *malloc* and *custom_malloc* have similar performance. On the Intel Phi, however, the system *malloc* outperforms the *custom_malloc* (independent of the number of regions). We experimentally verified (by forcing locks also on system malloc calls) that this inefficiency is not due to the synchronization overhead. We believe that pre-allocating a large memory buffer on the Intel Phi may affect the performance in two ways. First, the Intel Phi operating system performs lazy memory allocation (i.e., it progressively allocates memory as needed). This mechanism may make large allocations costly. Second, large pre-allocations may have bad interference with caching.

GPU implementation: The GPU implementation of the

above memory management mechanism requires addressing two issues: using a deadlock-free locking mechanism and reducing the synchronization cost. Ramamurthy [20] pointed out that, due to the SIMT architecture and the unfairness of GPU warp schedulers, common spin-lock implementations based on atomic *compare_and_swap* may cause deadlock. To avoid this problem, we use the deadlock-free implementation described in [21]. To reduce the synchronization cost on GPU, we associate a buffer region to each thread-block, thus limiting contention to threads belonging to the same block. This design also allows storing handles in shared memory for fast access. However, atomics on shared memory generally result in serialization and are costly [22]. We experimental verified that storing these handles in global memory leads to better performances. Finally, we note that the blocks are stored in global memory, and can therefore be accessed by all threads. In summary, each thread-block can *allocate* data only in its assigned regions, but can access data located in regions mapped to other thread-blocks.

VII. PERFORMANCE EVALUATION

In this section we evaluate the performance of the code generated by GRapid on three platforms: an 8-core Xeon E5-2609 CPU, an NVIDIA Tesla C2075 GPU, and an Intel Xeon Phi 5110P. This process will provide guidelines on the mapping of applications onto these platforms. We compiled CPU and Phi code through the Intel C++ compiler (icpc 13.1.2) and used the CUDA 5 toolkit to compile and run the GPU code. Data transfer times are included in the coprocessor results. In all cases, we show the speedup reported by the parallel code over a serial code running on a Xeon E5 processor.

To cover all the categories of graph applications, we implement BFS, A-DFA, PageRank and DFA Construction using GRapid. For BFS and PageRank, we used datasets from DIMACS competitions and Stanford-Large-Data Collection. The largest graph has 4M nodes and 34.5M edges. For A-DFA and DFA construction, we use DFA graphs with 30k-700k nodes and 70M-180M edges (typical sizes in regex matching paper [17]). Due to limited space, we omit the details of experiments with BFS.

A. A-DFA

Figure 7 and 8 report the performance of A-DFA. The datasets are DFAs with number of states varying from 30k to 400k, as reported on the x-axes. We recall that A-DFA makes the default transition of each DFA state n point backward to the state that has the highest number of transitions in common with n. The distance parameter d affects the amount of work (lines 3-10 of Figure 3): for larger d, the algorithm performs more state comparisons (and memory accesses). In the GPU case, we use three of the code variants and also show the performance reported by BFS. In the Intel Phi case, we use a bitmap-based working set and a variable number of threads.

We note the following observations. First, due to the irregularity of the work performed, the speedup of A-DFA on GPU is substantially inferior to that of BFS using the same code variants. Second, the GPU speedup of A-DFA decreases when the amount of work increases (that is, for larger d and datasets). Third, the relative performance of the code variants differs between BFS and A-DFA. Fourth, due to the more general purpose nature of its hardware, the behavior of the Intel Phi differs from the GPU. In fact, the speedup of A-DFA on the Intel Phi is far greater than that of BFS and increases with the amount of work (that is, with increasing d) from 9~15x to 43~67x. On the other hand, the speedup does not scale with the dataset size. In fact, larger graphs put more pressure on the cache, thereby limiting the performance. In addition, in contrast to BFS, A-DFA can effectively leverage all 60 cores available on the Intel device: the performance scales almost linearly until 120 threads (e.g., two threads per core). However, using all four hardware threads in each core does not provide further benefits. Finally, due to the complexity of the work, the multi-threaded CPU implementation of A-DFA achieves ideal speedup: roughly 8x on 8 cores.

B. PageRank

Figure 9 shows the speedup (and, for values < 0, the slowdown) of PageRank on dynamic datasets when using our hierarchical-CSR over performing a full CSR rebuild. Different provision methods are used: *0 provision* (reserve no space), *even provision* (reserve same space for each node) and *ratio provision* (reserve different space according to outdegree). The experiments are conducted on the Google weblink graph, which consists of 0.7M nodes and 2.5M edges. For the Even Provision, we use half of average outdegree as the default parameter. For the Ratio Provision, the parameter is 50% (reserve 50%*outdegree blank slots). We dynamically add an increasing number of nodes and edges (even distribution) to the original graph (x-axis). While doing so, we keep the average outdegree unchanged: the number of added edges is roughly 7 times that of added nodes. In the hierarchical-CSR implementation the PageRank kernel is slightly more complex because it must handle the hierarchical graph data structure. However, the static approach requires a full rebuild of the CSR representation and a large data transfer between the CPU and the coprocessor. As can be seen, since the amount of added

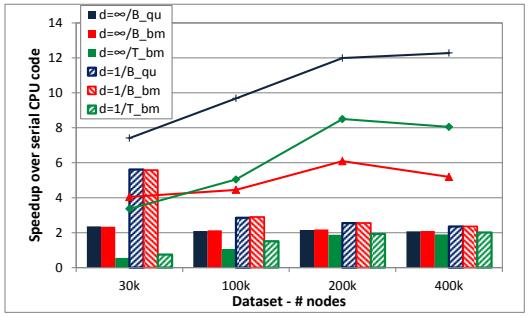

Fig. 7. A-DFA compression - Speedup of GPU over serial CPU implementation.

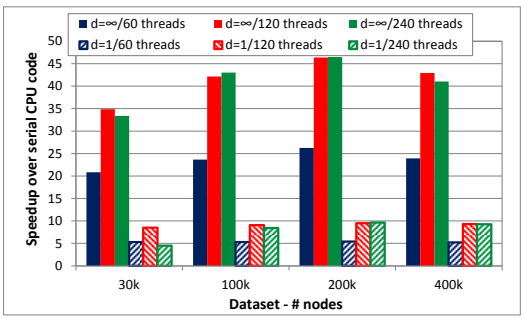

Fig. 8. A-DFA compression - Speedup of Intel Phi over serial CPU implementation.

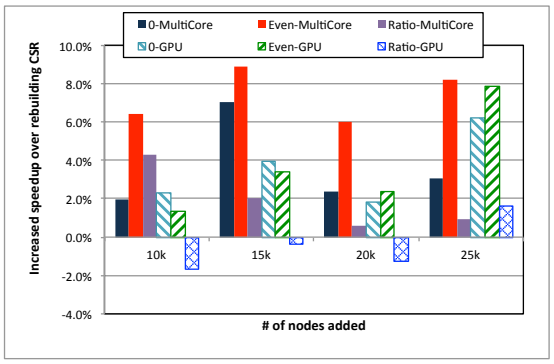

Fig. 9. PageRank on dynamic graphs – Speedup of hierarchical CSR over a full CSR rebuild on multi-core CPU and GPU (various provision conditions).

nodes is small (< 3.5%), the incremental, hierarchical approach is preferable to a full rebuild on both CPU and GPU for most of the cases. On CPU, even provision always get the best speedup. This can be explained by that the even provision benefits from the even distribution of the added edges. On GPU, however, the performance of 0 provision and even provision are vary close. Provision introduces blank slots in the CSR (level-1 node and level-1 edge arrays), which leads to underutilization of GPU hardware. Due to this reason, the ration provision downgrade the performance of GPUs since it leads to more unbalanced blank slots than even provision. We don't show the performance of the Intel Phi, because it is poor (always 10%~20% slow down). We believe that the memory allocation mechanisms within the Intel Phi's OS may interfere with our dynamic memory management scheme and cause this performance loss. We need to get more insights on the operation of the Intel Phi OS to improve the performance of our dynamic memory management scheme.

C. *DFA Construction*

Figure 10 shows the speedup of DFA construction on multi-core CPU, GPU and Intel Phi over a serial CPU implementation. We tested two code variants: one using coarse-grained and the other using fine-grained synchronization. We recall that the coarse-grained code variant has a critical section around the *subset.update* primitive, while the fine-grained version includes a thread-safe *subset* implementation that associates a fine-grained lock to each node of the double linked-list. On GPU, we use block-based mapping. In the kernel configuration, we set the number of blocks to twice the number of *SM* on the GPU and the block size to 128 threads. For the Intel Phi, we show the thread configuration reporting the best performance (the corresponding number of threads is indicated on top of the bars). We test DFA construction on four datasets with increasing size (x-axis). The multi-core CPU and the Intel Phi report the best and worst performance, respectively. Interestingly, the two code variants show comparable performance on all platforms (recall that the coarse-grained version uses the high-level *critical* primitive and requires minimal programming effort). The GPU and Intel Phi are consistently slower than the 8-threaded CPU code; the performance gap between the multi-core CPU and the many-core devices, however, decreases as the dataset size (and the runtime parallelism) increases. The GPU performance suffers from the irregular memory access patterns, cost of locking and branch divergence. The modest performance of the Intel Phi is in part due to the less efficient memory management module within the thin-OS running on this coprocessor.

Halloc is a state-of-the-art dynamic memory allocator for Nvidia Kepler GPU. Figure 11 shows the performance comparison between our memory allocator and Halloc in DFA construction on both Fermi (C2075) and Kepler (K20) GPU. Since the Halloc utilizes "__shfl" instruction supported only on Kepler, we modify the source code and port Halloc to Fermi GPU. "__shfl" instruction is used to distribute register values from one thread to the remaining threads in a warp. On Fermi, we let the lead thread to write the register value to shared/global memory first and then let the remaining threads in the same warp to read. Since this workaround is not efficient (more instructions and synchronization), the execution time of Halloc versioin on Fermi is roughly 3x of the version using our allocator. On the K20 GPU, the resuls show that our allocator achieves a 21%, 25%, 28% performance improvement over halloc on 60k, 280k, 700k, node DFA. Our allocator separates the permenant allocation from temparary allocation by utilizing the compile-time analysis so that deallocation and free memory space tracking is much simpler than the Halloc.

Table I summarizes the characteristics of the four applications (BFS, A-DFA, PageRank, DFA Construction) and the speedup reported on multi- and many-core platforms. In the 2nd column we indicate whether the graph topology is static or dynamic, and, in the latter case, whether it is modified by the application (*read-write* applications) or by external intervention (*read-only* applications). The 3rd column shows the application-specific attributes. The 4th column reports an indication of the complexity of the parallel work and its arithmetic intensity. As can be seen, many-core platforms outperform multi-core CPUs on static datasets. The Intel Phi is preferable to GPU for more complex computational patterns,

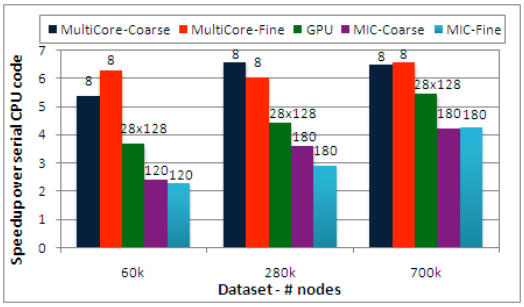

Fig. 10. DFA construction - Speedup of multi-core CPU, GPU and Intel Phi over serial CPU code (numbers of threads run are on top of the bars).

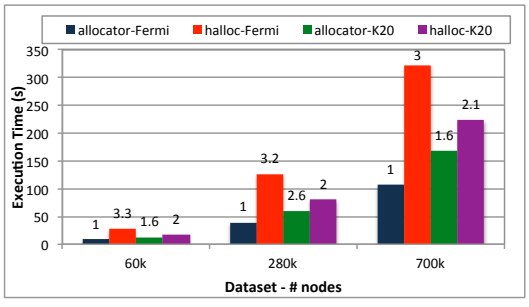

Fig. 11. DFA construction with different allocator on Fermi and Kepler GPU–numbers on the top of the bars indicate the speedup over the allocator-

TABLE I: Summary of applications and speedup over serial code.

Application	Application & Graph Type	Attributes	Comp. pattern & arith. intensity	Best Speedup		
				m-CPU	GPU	Phi
BFS	read-only static	Level (int)	Set level (simple& low)	2.5x	50x	3.5x
PageRank	read-only static/dynamic	Rank (double)	Calculate Rank (intermediate & intermediate)	6.3x	6.5x	9x
A-DFA	read-only static	Default Trans (int)	Compare trans. (complex, low)	8x	6x	45x
DFA construction	read-write dynamic	Trans Table (int)	Compare Subset (complex, low)	6.5x	5.5x	4.3x

whereas the arithmetic intensity is not a big discriminating factor. The three platforms report similar speedups on DFA construction; however, the multi-core CPU is in this case a slightly better choice, due to the presence of frequent dynamic memory allocation and synchronization. We note that manually generating code for the three considered devices would be a daunting and time consuming task: by automatically generating different versions of the code, GRapid allows the programmer to quickly identify the platform most suited to his application.

D. Comparison with Existing Codes

We evaluated our generated multi-core code against Green-Marl [14] and our GPU code against hand-tuned implementations. Currently very little Intel Phi code is available for benchmarking.

Existing programming model and compiler toolchains – We considered Green-Marl's BFS code and implemented A-DFA and PageRank using this system. Green-Marl does not support dynamic datasets and generates only CPU code. Our multi-core performances are about the same as those of Green-Marl's codes for all datasets.

Hand-optimized code – ***BFS***: In our previous work [13] we have shown that the adaptive selection of the considered GPU code variants leads to performance comparable to that of the hand-tuned BFS implementation by Merrill et al. [6].

A-DFA: We hand-coded an A-DFA implementation for GPU that reorganizes the memory accesses so to maximize locality and uses shared memory for faster access. However, especially on large datasets, the performance obtained by this implementation is no better than that reported by running our compiler-generated code with massive multi-threading.

PageRank: Our GPU-over-CPU speedup is about half of that reported by Duong et al. [23], that uses single-precision floating point arithmetic. We use double-precision arithmetic.

DFA Construction: The performance of the implementation based on coarse-grained locking is comparable to (and sometimes better than) that of a code that uses a custom *subset* implementation with fine-grained locking. This result holds on all considered platforms.

VIII. RELATED WORK

Parallel BGL [24], ParGraph [25], STAPL [26] and GraphLab[27] are extensible parallel graph libraries for multi-core CPUs and distributed systems. GraphChi [28] efficiently computes large graphs on a single CPU node. Green-Marl [14] offers a domain-specific language for graph analytics. The Galois system [15, 29] includes a programming model and a runtime component to dynamically extract parallelism from irregular applications by leveraging speculative parallelization. In this work, we target many-core processors. Differently from Green-Marl, we address also graph algorithms with dynamic data structures. In addition, unlike Galois's, our runtime does not offer support for speculative execution. Instead, we require the programmer to expose parallelism using platform-independent parallel constructs.

A few proposals have accelerated specific graph algorithms on NVIDIA GPUs [4-11]. Harish et al. [4] were the first to perform this operation. Better results have been reported in subsequent efforts, which covered a variety of algorithms (BFS [2, 5-9], single-source shortest path [7-9], minimum spanning tree [9], Delaunay mesh refinement [8, 9], points-to analysis [8-10], strongly connected components [11] and survey propagation [8, 9]). Among these, Hong et al. [7] proposed a virtual warp-centric programming model with more general applicability. In addition, Burtscher et al. [8, 9] identified computational patterns common to various graph applications and lessons learned from their implementation. We leverage these proposals in two ways. First, we benefit from the lessons learned from these efforts in the code generation phase. Second, these implementations provide a library of reusable algorithmic patterns (e.g., BFS) and mechanisms (e.g., synchronization).

Mapgraph [30], VertexAPI2 [31] and Gunrock [32] are GPU-targeting tools for graph analytics, not compiler frameworks to generate multiple GPU code variants for generic graph. They provide library implementations of specific graph algorithms and APIs to customize graph analytics. Users still require hand-coding in CUDA/OpenCL and implementing predefined callback functions (gather, scatter).

TOTEM [2] and Medusa [3] are more general programming frameworks for graph algorithms operating on static datasets (i.e., they do not include support for dynamic memory operations). Our work aims to automatically generate different code variants for GPU and Intel Phi devices starting *from a platform-agnostic interface* and to also support *dynamic* datasets. Unlike TOTEM, we do not consider graph partitioning across devices and cooperating CPU-GPU execution. Medusa also explores different graph storages (e.g., edge-oriented storage), graph-aware buffer schemes and multi-GPU execution. These mechanisms, however, can be incorporated in our framework.

Existing directive-based frameworks for GPU (OpenMPC [19], OpenACC [33], among others [34]) target generic applications and do not include graph-specific data structures and optimizations.

IX. CONCLUSION & FUTURE WORK

We have proposed a system for rapid deployment of graph applications on multi-core and many-core platforms. Our future work will span three directions. First, by leveraging open-source frameworks such as Cetus [3], we will improve the compiler so to automatically extract parallelism from the application. Second, we will extend the runtime system with online selection of code variants. Finally, we will validate our toolchain on more graph applications.

X. REFERENCES

[1] Nvidia. "Popular GPU-Accelerated Applications Catalog," http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf.

[2] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest," in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[3] J. Zhong, and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE Trans on Parallel and Distributed Systems*, 2013.

[4] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in Proceedings of the 14th international conference on High performance computing, Goa, India, 2007, pp. 197-208.

[5] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in Proceedings of the 47th Design Automation Conference, Anaheim, California, 2010, pp. 52-55.

[6] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, 2012, pp. 117-128.

[7] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, San Antonio, TX, USA, 2011, pp. 267-276.

[8] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2012, pp. 141-151.

[9] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus Topology-driven Irregular Computations on GPUs," in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[10] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, 2012, pp. 107-116.

[11] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing Strongly Connected Components in Parallel on CUDA," in Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011, pp. 544-555.

[12] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM,* vol. 33, no. 8, pp. 103-111, 1990.

[13] D. Li, and M. Becchi, "Deploying Graph Algorithms on GPUs: an Adaptive Solution," in Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, 2013.

[14] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: a DSL for easy and efficient graph analysis," in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, England, UK, 2012, pp. 349-362.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, San Diego, California, USA, 2007, pp. 211-222.

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.: Addison Wesley, 2006.

[17] M. Becchi, and P. Crowley, "A-DFA: a Low-complexity Compression Algorithm for Efficiant Regular Expression Evaluation," *ACM Trans. Archit. Code Optim.,* vol. 10, no. 1, 2013.

[18] Nvidia. "Fermi Compute Architecture White Paper," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[19] S. Lee, and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.

[20] A. Ramamurthy, "Towards Scalar Synchronization in SIMT Architectures," Electrical and Computer Engineering, The University of British Columbia, 2011.

[21] Sarnath. "Spinlock on a GPU," http://forums.nvidia.com/index.php?showtopic=98444&st=40/.

[22] Harrism. "Atomic Operations on GPU," http://stackoverflow.com/questions/3148345/are-atomic-operations-on-global-memory-in-cuda-performed-in-parallel-across-a-wa.

[23] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen, "Parallel PageRank computation using GPUs." pp. 223-230.

[24] D. Gregor, and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," in Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA, 2005, pp. 423-437.

[25] F. Hielscher, and P. Gottschling. "ParGraph," http://pargraph.sourceforge.net.

[26] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: an adaptive, generic parallel C++ library," in Proceedings of the 14th international conference on Languages and compilers for parallel computing, Cumberland Falls, KY, USA, 2003, pp. 193-208.

[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud." pp. 716-727.

[28] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in Processings of the 10th USENIX conference on Operating

Systems Design and Implementation, Berkeley, CA, USA, 2012, pp. 31-46.

[29] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-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, San Jose, California, USA, 2011, pp. 12-25.

[30] "Mapgraph: GPU-accelerated Graph Analytics," June 22, 2014; http://www.systap.com/mapgraph.

[31] "VertexAPI2 - Large-Graph Analytics on GPUs," June 23, 2014; http://www.royal-caliber.com/resources.html.

[32] "Gunrock: High-performance Graph Primitives on GPU," June 23, 2014; http://gunrock.github.io/gunrock/.

[33] OpenACC. "OpenACC: Directives for Accelerators," April 17, 2014; http://www.openacc-standard.org.

[34] S. Lee, and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah, 2012, pp. 1-11.