LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs

G. Shashidhar, Rupesh Nasre

IIT Madras, India
{shashi, rupesh}@cse.iitm.ac.in

Abstract. We propose LightHouse, a GPU code-generator for a graph language named Green-Marl for which a multicore CPU backend already exists. This allows a user to seamlessly generate both the multicore as well as the GPU backends from the same specification of a graph algorithm. This restriction of not modifying the language poses several challenges as we work with an existing abstract syntax tree of the language, which is not tailored to GPUs. LightHouse overcomes these challenges with various optimizations such as reducing the number of atomics and collapsing loops. We illustrate its effectiveness by generating efficient CUDA codes for four graph analytic algorithms, and comparing performance against their multicore OpenMP versions generated by Green-Marl. In particular, our generated CUDA code performs comparable to 4 to 64-threaded OpenMP versions for different algorithms.

1 Introduction

Processing big graphs in a reasonable time requires huge computing power as well as ability to perform operations in parallel. Unfortunately, graph algorithms are notoriously difficult to optimize and parallelize. The main source of difficulty in graph algorithms stems from a technicality called *irregularity*. Graph algorithms are irregular because their memory access, control-flow and communication patterns cannot be predicted at compile time (as they depend upon the nature of the input graph, which is unavailable during compilation).

In the last decade, we made a substantial progress in understanding graphs and their access patterns in various algorithms. It has been shown that graph algorithms indeed have a good amount of parallelism [8]. However, the analysis and the parallelization techniques developed for *regular* programs (such as dense matrix algebra) need not be best suited for graph-based computation [15]. Graph algorithms are more amenable to dynamic processing, rather than compile-time static processing performed for regular programs.

Over the years, researchers have optimized parallel graph processing for multi-cores [10,9,19], GPUs [3,13], CPU clusters [1,12], and for heterogeneous combination of these [5]. However, several of these codes can only be used and modified by experts alone. Domain experts from various fields such as astronomy, physics, chemistry and biological sciences, who are not experts in high-performance computing, often cannot directly utilize the proposed techniques.

One of the interesting approaches to allow non-experts to program in a domain is using domain-specific languages (DSLs). DSLs have been quite successful in various fields, such as matrix computations using MATLAB, string processing using regular expressions, and statistical processing using R. In a similar spirit, DSLs have been developed for graph algorithms with a hope for non-experts to achieve reasonable performance without worrying about the intricacies of the hardware platform or parallel execution. Unfortunately, graph DSLs are currently limited to one type of platform. For instance, a graph DSL Green-Marl [6] has a backend to generate code for multi-core CPUs, but is unsuitable for GPUs. Efficient code-generation for GPUs is challenging due to separate memories of CPUs and discrete GPUs (variables need to be defined, copied and accessed appropriately in the generated code), GPUs being more suitable for hierarchical computation spanning individual thread, warps, thread-blocks and GPU threads (the compiler should be able to identify scenarios where such a hierarchical code can be generated), lack of logical locks (which are routine in CPU libraries), and generating code for various data structures using arrays and offsets rather than pointers. We highlight and address these challenges in this work. Following are our main contributions.

- We create a GPU backend for a graph DSL. In this process, we exploit various architectural features of the GPU, and develop techniques to map the high-level language constructs to efficient backend processing. While we use CUDA as the target language, the techniques developed are general enough to be applicable to other GPU languages as well.
- To reduce the learning curve for a programmer, we use the language specification of an existing DSL called Green-Marl, instead of developing a new language. Green-Marl already has an OpenMP backend for multi-core execution. This also provides us with an opportunity to compare the efficiency of `LightHouse`-generated GPU code with a well-optimized CPU backend.
- We overcome several GPU-centric challenges (separate memories, hierarchical computation, SIMD execution, etc.) by optimizing the abstract syntax tree, and illustrate the efficacy of our compiler by generating four graph algorithms: computing bipartite matchings, finding single-source shortest paths, computing page-rank, and calculating conductance of a graph. Our experimental evaluation reveals that the performance of the generated CUDA code considerably varies compared to that of the multi-core CPU version (comparable to 4 to 64-threaded OMP version), but overall, provides a productive way to generate code for GPUs.[1]

2 Green-Marl Language Specification

In this section we introduce the constructs of the Green-Marl language. Green-Marl has constructs that can be used to describe many graph analytic algorithms. The language does not allow graph mutation, that is, the graphs are static. It

[1] `LightHouse` code is available at `http://pace.cse.iitm.ac.in/tools.php`.

supports basic types such as nodes and edges as well as operations on collections (such as a set of nodes or a sequence).

Algorithms in Green-Marl have a single procedure with input graph as argument along with the properties defined on the nodes and the edges of the graph. The procedure returns a value or a property. The basic data types such as `int`, `bool`, `float` are supported as property types. Nodes and Edges are also supported as basic collection types in Green-Marl. To access individual elements in the collections, Green-Marl supports iterators. In particular, it provides node and edge iterators to navigate the graph. The order in which the graph elements are traversed is decided by the collection type (a set or a sequence).

```
1  Procedure triangle_counting(G: Graph): Long // Return value type
2  {
3      Long T;
4      Foreach(v: G.Nodes)
5        Foreach(u: v.Nbrs) (u > v) {
6          Foreach(w: v.Nbrs) (w > u) {
7            If ((w.HasEdgeTo(u)))
8              T += 1;
9            }
10         }
11       }
12     }
13     Return T;
14 }
```

One of the advantages of the Green-Marl syntax is that most of the code is sequential, which is very intuitive for the programmer. Parallelism is implicitly specified using a `foreach` construct. Combined with iterators, the `foreach` loop allows a compiler to assign tasks to different processing workers (iterations mapped to threads). Green-Marl follow the fork-join style of parallel execution.

At line 4 of `triangle_counting` procedure, a set of parallel executions is created starting the execution of the loop-body. At line 5, each running parallel execution creates more parallel executions and waits for their completion at line 11. Each of the outer parallel executions continues after line 11 and exits at line 12. The scope of the iterators used inside a foreach statement is only within the statement body.

The parallel execution style of Green-Marl has data races on the location read from and written to concurrently. Green-Marl provides *reduction* statements to provide determinism on some operations.

```
1    reducedValue  += expr;
```

`expr` values computed by all the parallel executions are reduced to `reducedValue` such that the result would be the same as computed sequentially. The reduction operation can be +, *, min, max, bitwise AND and OR. `reducedValue` should be read only after all the parallel executions have finished the execution of the reduction statement. `Node` and `Edge` properties can also be reduced.

```
1  Foreach(n: G.Nodes)
2      Foreach(t: n.Nbrs)
3          n.A += t.B;
```

The property `B` is reduced into property `A`. The frontend of the Green-Marl provides syntax checking to identify any conflicts in the locations being read in `expr` and written to in `reducedValue`. In addition to the normal reduction

statement, Green-Marl provides constructs to gather values in the context which minimized or maximized the expression.

Output of the Green-Marl compiler *gm_comp* is a C++ code annotated with OpenMP pragmas. This code needs to be compiled with another code containing the `main` entry point to generate the final application.

Green-Marl Frontend: The front end provides the syntax checks and parallel semantics checks, and generates an Abstract Syntax Tree (AST). The higher level description of the program helps in identifying possible problems in the parallel program semantics like data-races. For instance, consider this code:

```
1  Node_Prop<Int> A; // node property
2  Foreach(n: G.Nodes)
3    Foreach(t: n.Nbrs)
4      n.A = t.A;
```

At line 4, the iterators `t` and `n` are used to update the node property `A`. The property `A` is written through iterator `n` and read through iterator `t`. At this point, there is no guarantee that `n` and `t` could not create a data conflict on `A`; that is, `n` in one thread and `t` in another may refer to the same node leading to a race. The frontend analysis finds that at line 3, iterator `t` is defined on `n`'s neighbors. The analysis reduces iterator `t` to random access along `n`. At this point there is a write by `n` and the random access read by reduction from `t`. A data conflict exists between iterator `t` and `n` on the node property `A`. The compiler issues errors on identifying such conflicts. After parsing and checking of the input specification, the front end generates an AST representing various constructs defined in the Green-Marl language.

Green-Marl Optimizations: A set of architecture independent transformations is applied on the AST: (i) Perform loop fusion which combines two `foreach` loops that have the same type of iterator and no loop-carried dependence. (ii) Combine assignments that are running on the same iterator type into a single parallel loop. (iii) Hoist the temporary property definition out of the sequential loop to save the repeated allocations and deallocations. (iv) Convert the reduction inside a sequential loop to a normal assignment. (v) Move a reduction to the outermost parallel loop just after the definition / declaration of the target symbol. If there is no such loop then the compiler converts the reduction to a normal assignment. The output of this phase is a modified AST which is transformed by the above mentioned optimizations.

Green-Marl Backend: The existing backend currently generates OpenMP code for multi-core CPU processing. The backend traverses the AST and generates `parallel for` construct for the outermost `foreach` loop. For single value reductions, `atomic` construct is utilized, while for multi-value assignments, a lock-based code gets generated. Note that generating such as a lock-based code for GPUs is not an option due to inefficient execution of locks in the presence of hundreds of thousands of threads. Further, reductions on GPUs can be accomplished by a hierarchical computation across warps and threads-blocks.

```
1  Procedure Test (G: Graph,
2      A: N_P<Int>, root: Node) {
3
4      N_P<Int> B;
5      Int rootValue;
6      Foreach (n: G.Nodes) {
7          Foreach (s: n.Nbrs) {
8              n.B = n.A + s.A;
9          }
10     }
11     rootValue = root.B;
12 }
```

Fig. 1: Green-Marl Example

Symbol	Type	Parent	Allocate in
G	Graph		GPU
A	NP< *Int* >		GPU
B	NP< *Int* >		GPU
n	Node::I	G	GPU
s	Node::I	n → G	GPU
root	Node		CPU
rootValue	Int		CPU

Fig. 2: Symbol Table for the Program in Figure 1

This demands careful management of cooperation across threads. Finally, the generated C++ procedure may contain temporary as well as global variables. Temporaries get converted to thread-local variables, while global variables can be directly accessed by OpenMP threads. However, in CUDA, the globals from CPU are not directly accessible on GPUs (unless unified memory is used for storing data). This demands identifying the locations of variables' access as well as their definitions. If the two devices are different, the compiler needs to insert code to explicitly transfer such variables across the two devices.

3 GPU Code Generation

This section presents the challenges that `LightHouse` faces for efficient GPU code-generation of the Green-Marl language specification. Apart from translating the usual constructs, `LightHouse` primarily involves four subtasks, which we discuss in the following subsections.

3.1 Identifying Parallel Regions

This phase selects the part of the code to be run on the GPU. The `Foreach` construct specifies parallelism implicitly. `LightHouse` generates a kernel corresponding to the parallel loop. Only the outermost `Foreach` is selected to be run on the GPU in parallel. For instance, for the code shown in Figure 1, the outer `Foreach` on line 6 gets converted to a kernel which contains the body of the loop. Thus, loop iterations are executed by concurrently running threads. The inner `Foreach` on line 7, on the other hand, gets compiled into a sequential `for` loop executed by each thread within its kernel code.

3.2 Identifying Variable Location

Unlike in the CPU backend, `LightHouse` needs to identify the variable location (whether on CPU or GPU). This is decided by a static pass which relies on a *use-def* analysis to find out the variables read and written to at different instructions

in the program. `LightHouse` maintains a symbol table which is populated with variables and their type information. All the variables inside the parallel regions have to be accessible to the GPU. These variables are allocated in the GPU (global) memory. Variables of primitive data types can be passed as parameters to the kernel. Variables that need to be in the GPU memory are marked to have a *GPU Scope*. In addition, a variable written in the GPU kernel but used in the CPU code needs to be transferred to the CPU. For instance, Table 2 shows the symbol table for the Green-Marl program in Figure 1. The variables accessed inside the foreach loop at line `6` need to be accessed in the GPU. This includes Graph `G`, Node properties `A` and `B`, and Node iterators `n` and `s`. The iterator `n` traverses all the nodes of `G`, and `s` traverses the neighbors of those nodes.

Each outermost `foreach` loop defines a new *scope* for the GPU. All the variables accessed in the `foreach` loop have to be declared and defined inside the GPU scope. Node and Edge properties are converted to arrays. These arrays are allocated space in GPU's global memory and are sent as kernel launch parameters. Temporary variables used inside the foreach loops are declared inside the kernel and are added to the lexicographic scope of the kernel. Variables which are defined and used outside the kernel are added to the CPU (Global) scope.

For the variables in the Global scope that are also accessed inside the foreach loop, `LightHouse` creates a copy in the GPU Global memory. It modifies the variable accesses inside the kernel to the corresponding copies in the GPU scope.

3.3 Generating Indices for Memory Accesses

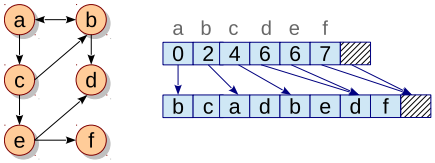


Fig. 3: Graph in CSR Format

The input graph is stored in Compressed Sparse Row Format (CSR) which consists of two arrays (another array for weights). The row array R has the adjacency list of all the nodes in the graph. The column array C has the indices into the row array for the starting index of the adjacency list for each node. Figure 3 shows an example graph in the CSR format. For each iterator the index pattern based on the CSR format needs to be generated for CUDA threads. The parent information of the iterator in the symbol table is used to generate the index values.

For instance, consider the code snippet: **Foreach** (n: G.**Nodes**) ... In this code, `n` is an iterator on all the nodes in the graph. Running this `foreach` loop in a fully parallel manner on GPU assigns one node to each CUDA thread. The corresponding index pattern generated is as follows.

```
1  n = threadID;
2  if (n > numNodes)
3    return;
```

Here `n` is the node id being processed by a thread, derived from the unique id `threadID` of the thread, computed in CUDA as `blockIdx.x * blockDim.x + threadIdx.x`. Similarly for the pattern below,

```
1  Foreach (n: G.Nodes)
2    Foreach (s: n.Nbrs)
3      ...
```

the inner `foreach` loop is converted into a sequential loop. Iterator `s` goes over all the neighbors of `n`. The generated code is:

```
1  n = threadID;
2  if (n > numNodes)
3    return;
4  for (i = C[n]; i < C[n+1]; i++) {
5    s = R[i];
6    ...
```

Similar patterns are defined for in / out neighbors and in / out edge iterators.

3.4 Generating Code for Reduction Statements

Green-Marl provides min, max, add, mult, or, and, inc reduction constructs. `LightHouse` converts these reduction operations to atomic operations on GPU.

```
1  Int T = 0;
2  Node src, dst;
3  Foreach (s: G.Nodes)
4    Foreach (t: s.Nbrs)
5      T<src, dst> max= s.A + t.A<s, t>;
```

The assignment at line 5 performs a max-reduction of the expression `s.A + t.A` to variable `T` and assigns the corresponding node ids to `src` and `dst`.

```
1  int T = 0;
2  GPUMemCpy(GPU_T, T, HostToDevice);
3  KernelCall<<<LaunchPara>>>(C, R, A);
4  GPUMemCpy(from, GPU_from, DeviceToHost);
5  GPUMemCpy(to, GPU_to, DeviceToHost);
6
7
8  KernelCall(C, R, A) {
9    s = threadID;
10   if(s > NumNodes)
11     return;
12   for(i = C[n]; i < C[n + 1]; i++) {
13     t = R[i];
14     expr = s.A + t.A;
15     atomicMax(&GPU_T, expr);
16     if(localExpr < expr) {
17       localExpr = expr;
18       localFrom = s;
19       localTo = t;
20     }
21   }
22   SoftwareBarrier();
23   if(localExpr == GPU_T)
24     chooseThread = threadID;
25   SoftwareBarrier();
26   if(chooseThread == threadID) {
27     GPU_from = localFrom;
28     GPU_to = localTo;
29   }
30 }
```

Fig. 4: Code for multiple atomic assignment

The language definition of Green-Marl demands evaluation of line 5 in an atomic manner. That is, assignments to `T`, `src` and `dst` must be seen by other threads as happening together. We call its type as *multiple atomic assignment statement*. On CPUs, the Green-Marl OpenMP backend uses logical locks to implement multiple atomic assignments. CUDA neither has a support for such a statement type, nor is it feasible to use locks in the presence of hundreds of thousands of concurrent threads. For efficiency, we implement such reductions using software barrier and atomics. Figure 4 shows the CUDA code generated for *multiple atomic assignment statement*. All the

threads store the expression to be reduced (line 14) and perform an atomic minimum (or maximum) on the target global value (line 15). If the new expression value is lesser (respectively, larger) than the previously computed value, then the values of the sub-expressions are stored into local variables (line 17 - 19). All the threads synchronize at the end of the `foreach` loop (line 22) and compare their individual local copies of reduction expressions with the reduced value. Threads having the same value are the potential threads which might have written the reduced value. According to the language semantics, one of these potential threads must assign the reduced value. All the potential threads write their `threadID` to a unique location (line 24) and only one of the writes gets reflected at the end. This thread is chosen to write its value of local sub-expressions to the global value (lines 26–28). This ensures that only one of the potential threads which had reduced the value to its minimum / maximum also writes the corresponding sub-expressions to the global location.

Lines 22 and 25 use a call to `SoftwareBarrier()` which implements a global barrier across all the threads on the GPU. A global barrier is a synchronization primitive that guarantees that all threads from all the thread-blocks belonging to a kernel reach a specific point in the code before any thread may progress beyond that point. CUDA supports a barrier at the thread-block level (`syncthreads`). However, a global barrier (across thread blocks) needs to be emulated in software. We implement it without using atomics [13,21].

4 Program Optimizations

In this section we present three important GPU-specific optimizations that are implemented in `LightHouse` to improve the performance of the generated CUDA code. The optimizations work with the control-flow graph and the def-use chains.

4.1 Eliminating Atomics

Reduction of a boolean value can be implemented without using atomics by initializing a value to the reduction variable and set the variable based on the condition. As only one thread is enough to change the value of the reduction variable, subsequent reduction does not change the semantics of the program.

`atomicOr`(&A, val);	`atomicAnd`(&A, val);
becomes	**becomes**
// initialized outside kernel	// initialized outside kernel
A = false;	A = true;
....	....
if (val) A = `true`;	if (!val) A = `false`;

In the above translated code, there is a data race on `A`, but the threads participating in the race set `A` to the same value. So, the race is benign. In case of multiple assignment statement, a sub-expression of type boolean can be assigned similar to the above code without using the software barriers. This gets rid of the limitation of the software barrier which demands all threads participating in the barrier to be resident (which reduces concurrency).

4.2 Loop Collapsing

Typical implementations of graph algorithms are *vertex-centric*, that is, a vertex is assigned to a thread and the thread operates on all its neighbors. When the input graph's degree-distribution is rather uniform, as in road networks, a vertex-centric algorithm assigns almost equal amount of work per thread. However, for a graph with skewed degree-distribution, as in social networks, a vertex-centric algorithm suffers from high load-imbalance [22,2]. The problem is exacerbated on GPUs as warp-threads execute in SIMD fashion. One way to remove the load imbalance is to make the algorithm *edge-centric*, that is, transform the traversal on neighbors of all the nodes to traversal on all the edges. One thread is assigned to work on one edge which creates evenly balanced workload and hence improved parallelism. In CSR representation of the graph, each thread accesses contiguous memory locations on the edge list. CUDA combines such contiguous memory accesses from a warp into a single global memory access (called as memory coalescing). This increases the memory bandwidth of the process and results in better performance.

```
Foreach(s:G.Nodes)                    Foreach(e:G.Edges) {
    Foreach(t:s.Nbrs)      becomes        s = e.FromNode();
        ...                               t = e.ToNode();
                                          ...
```

From Green-Marl language perspective, such a transformation can be depicted as shown on the left. In this code, *FromNode* and *ToNode* are API that return end-points of an edge. Such an approach needs an array of edges rather than the CSR format. However, converting a vertex-centric algorithm to an edge-centric version may change synchronization requirements. For instance, in a *pull-based* implementation a thread operating on a vertex reads-in attributes from its in-neighbors and updates the current vertex's attribute. In such an approach, each vertex is being written to by only one thread, and hence threads need not synchronize their writes. However, when such a pull-based implementation is combined with edge-centric version, single-writer guarantee cannot be enforced, necessitating synchronization. Typically, for simple attributes (such as distance of a vertex or pagerank value), an atomic instruction suffices for correct execution (e.g., atomicMin for the shortest paths computation).

4.3 Full Device Occupancy

We also studied the effect of occupancy in the context of graph algorithms, by generating codes with full-occupancy and otherwise. We observed in our experiments that although occupancy is useful, its effect is limited in the case of graph algorithms and gets overshadowed by other effects such as launch configuration, memory coalescing, and thread-divergence.

4.4 Limitations of `LightHouse`

Although our code generator is automated, it can be improved in multiple aspects, such as generating code for heterogeneous systems, supporting graph mu-

Graph	#Nodes (millions)	#Edges (millions)	OpenMP 1-thread(in msec)				
			MATCH	SSSP	COND	PAGERANK GATHER	PAGERANK PROPAGATE
Epinions	0.076	0.509	11	11	1	48	139
LiveJournal	4.848	68.994	1432	1347	50	11818	21119
Pokec	1.633	30.623	273	1073	16	6267	8563
Orkut	3.073	117.185	687	3779	46	10724	20945
USA	23.947	57.709	1705	>35min	125	14312	26886

Table 1: Benchmark graphs and baseline performance

tation (would need changes in the language), reducing synchronization among threads, and optimizations using GPU shared memory.

5 Experimental Evaluation

We added a CUDA backend to Green-Marl to read the AST and generate GPU code as detailed in the previous sections. Thus, for the same graph algorithm specification, we are now able to generate both OpenMP as well as CUDA codes. This allows us to faithfully compare the performances of the generated programs.

5.1 Experimental Setup

We generated CUDA and OpenMP codes for four graph analytic algorithms: bipartite matching (MATCH), single-source shortest paths (SSSP), page-rank (PAGERANK), and conductance (COND) of a graph. MATCH is a matching algorithm where a random edge is selected as matching between two nodes. The algorithm returns one of the maximal matching and not the maximum matching. Because of the randomness the algorithm can be run in parallel. SSSP computes the shortest paths in a directed graph from a designated source vertex, and uses a variant of Bellman-Ford algorithm. PAGERANK calculates the importance of each node in the graph using the following formula.

$$PageRank(n) = \frac{(1-d)}{NumNodes} + d * \sum_{t \epsilon IncomingNodes(n)} \frac{PageRank(t)}{OutDegree(t)} \quad (1)$$

COND identifies how *well-knit* a graph is based on the degree distribution. The four algorithms test various aspects of our code-generator: MATCH involves testing data parallelism, SSSP tests generation of multi-atomic assignment, PAGERANK tests floating-point operations, while COND tests reductions and conditional evaluation of expressions.

Table 1 shows the benchmark graphs used in our evaluation along with their sizes in terms of the number of nodes and number of edges. The sizes range from

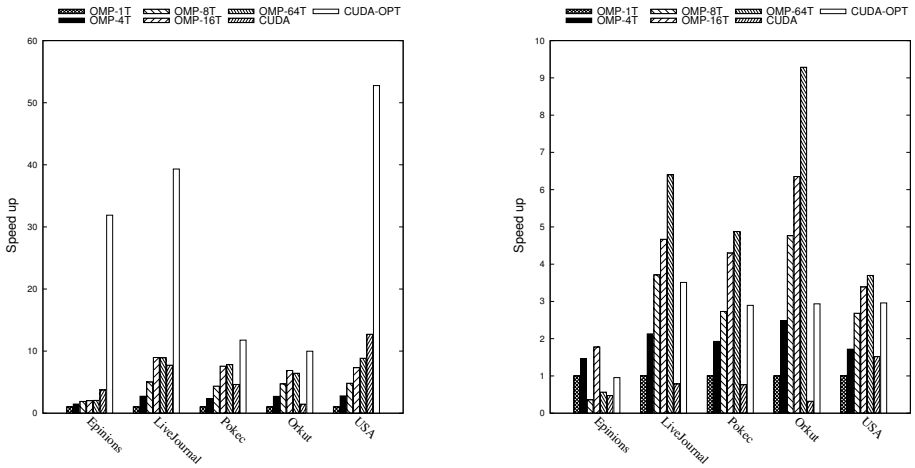
0.5 million edges (for Epinions) to 117 million edges (for orkut). All the graphs are obtained from SNAP [11]. The last columns of the table also show execution time of single-threaded OpenMP version for the three graph algorithms. We use it as a baseline for comparison of multi-threaded OpenMP and CUDA versions. We also compare our generated SSSP code against hand-optimized CUDA versions from LonestarGPU [3] and Totem [5]. Each algorithm implementation is run in CUDA and OpenMP frameworks with 1, 4, 8, 16 and 64 threads. The benchmarks for OpenMP are run on an Intel XeonE5-2650 v2 machine with 32 cores clocked at 2.6 GHz with 100 GB RAM, 32KB of L1 data cache, 256KB of L2 cache and 20MB of L3 cache. The machine runs CentOS 6.5 and 2.6.32-431 kernel, with GCC version 4.4.7 and OpenMP version 4.0. The CUDA code is run on Tesla K40C device which has 2880 cores clocked at 745 MHz with 12GB of global memory. The GPU device is connected to the same CPU device. CUDA_OPT is the baseline version with `Eliminating Atomics` and `Loop Collapsing` enabled. The execution time is taken after all the data necessary for computation is copied to respective memories till the procedure ends.

5.2 Experimental Results

Figure 5a shows the speedup obtained by the OpenMP and CUDA versions of **MATCH** compared to the single-threaded OpenMP version. We observe that CUDA_OPT considerably outperforms the OMP version's maximum performance. The algorithm has a nested `Foreach` loop which goes over neighbors of all nodes. CUDA_OPT converts this nested `Foreach` loop into a single `Foreach`-on-edge loop. Further, it converts the reduction of a boolean variable inside the nested `Foreach` loop to a normal assignment. Its high speedup is due to less conflicts across threads and load-balanced task distribution.

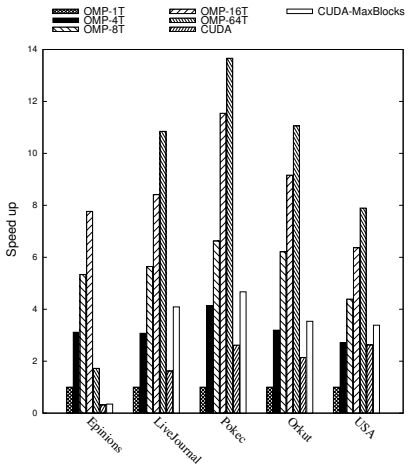
Figure 5b shows the results for **COND**. We observe that the OpenMP version performs considerably better and scales well, achieving a speedup of 9.3× for `orkut`. COND has atomics-based reductions to a variable from all the nodes of the graph which turn out to be slightly expensive in the presence of massive multithreading such as GPUs. Nonetheless, CUDA_OPT performs reasonably good and is comparable to 4-threaded OpenMP version.

PAGERANK is run with damping factor $d = 0.85$ and error tolerance of 0.0001 for maximum 40 iterations. It can be implemented both as a gathering or a propagating approach. In the former, every node gathers the pagerank of its incoming nodes to calculate its own pagerank. An advantage of gathering is that it does not need atomic writes, as every node is owned by a single thread. Figure 5c shows Pagerank results, which indicate that OpenMP scales well with number of threads. In case of CUDA, gather-based code improves synchronization, but also increases load-imbalance, as each thread needs to sequentially process all the incoming neighbors. On the other hand, in propagation-based code, every node propagates its pagerank to its outgoing nodes. The propagation demands atomics, but due to CUDA_OPT's node-based to edge-based optimization, the load-balance improves, leading to better performance, as shown in Figure 5d.

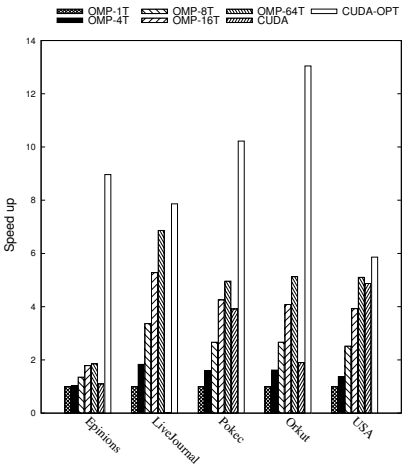


(a) MATCH

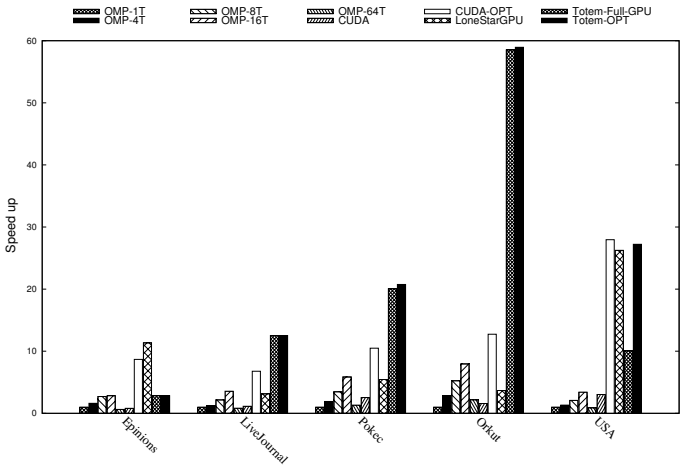
(b) COND



(c) PAGERANK-GATHER



(d) PAGERANK-PROPAGATE



(e) SSSP

Fig. 5: Performance of MATCH, SSSP, PAGERANK and COND on input graphs

Figure 5e shows the speedup obtained by the OpenMP and the CUDA versions of **SSSP** compared to the single-threaded OpenMP version. We observe that, in contrast to MATCH, the OpenMP version performs better in case of SSSP up to 16 threads (8× speedup). In comparison, our CUDA version performs consistently better for each graph. SSSP has a nested `Foreach` loop to propagate the distance value to all its neighbors. Along with minimizing the distance, each iteration marks the neighbors for propagation in the next iteration. Due to the irregular nature of the graphs and the algorithm, the number of conflicts on a memory location and load-imbalance increases with the number of threads. CUDA_OPT converts the traversal of neighbors of all nodes into edge-traversal which enables more parallelism. Totem's peak performance is achieved when all the graph nodes are processed on the GPU. It is hand-tuned to minimize the synchronization usage. In addition, automated code-generation of `LightHouse` has its bookkeeping overheads, which can be overcome by adding more architecture-independent optimizations to `LightHouse`.

Overall, we illustrate that `LightHouse` was able to generate well-performing CUDA versions from the same high-level description of the graph algorithms.

6 Related Work

Green-Marl [6] is a DSL for graph analytic algorithms running on shared memory systems. We explain Green-Marl's language features and code-generation briefly in Section 2. We use Green-Marl's specification as our language syntax. This allows us to retain existing productivity of the programmer. Further, in our experience, Green-Marl's syntax is intuitive (close to algorithmic description), well-defined and easy to learn; thus making it ideal for new domain experts.

Elixir [16] is a system for synthesizing irregular algorithms on multi-core platforms. Programmers specify the parallel computation as a set of *operators*, which is executed by multiple threads. Efficient execution of operators necessitates a good scheduling, which is often application dependent. Therefore, Elixir also provides a flexibility of specifying *schedules*, which could be customized as per the needs of an application. This allows generation of multiple implementations of the same algorithm (operator). Elixir also performs auto-inferencing to identify the next set of graph elements (nodes or edges) to be processed from the specification. An extension of Elixir [17] uses planning to generate schedules as well as synchronization automatically. Compared to Elixir, Green-Marl's syntax does not involve schedule specification and `LightHouse` targets GPUs which pose different challenges as discussed throughout the paper.

Halide [18] is a DSL for image processing. It provides a set of filters and a pipelined execution, where the output of one filter acts as input to the other. Users can write their own filters and alter the schedule to achieve the best results. Halide programs are restricted to stencils, in which the memory access pattern is *regular* (known at compile-time). Similar to our goal, Halide generates code for multiple platforms such as GPUs and heterogeneous CPU+GPU combination. `LightHouse` differs from the Halide compiler because the access patterns

of graph algorithms are *irregular*, requiring dynamic parallelization techniques. This means that the related optimizations in case of graph algorithms need to be deferred until run-time.

While there are only a few DSLs for irregular codes, there are several library-based platforms and parallelization frameworks proposed for processing graph algorithms. Galois [10] is a C++ framework for writing multi-core graph algorithms. A salient feature of Galois is that it supports morph algorithms also, wherein the graph structure changes. Ligra [19] is a framework for paralleliz-ing input-dependent programs, such as graph algorithms. LonestarGPU [3] and Totem [5] are frameworks for GPU and heterogeneous implementations of graph algorithms respectively. Medusa [24] is a C/C++ library-based approach to parallelize graph algorithms on multiple GPUs. GPU code generators for sparse matrix-vector multiplication [20] are also relevant.

Graph algorithms [10,15] have been shown to bear enough parallelism especially in the context of distributed [2,4,14] and heterogeneous systems [5]. G-Streamline is a software-based runtime approach to eliminate control-flow and memory-access irregularities from GPU programs [23]. DyManD is an automatic runtime system for managing recursive data structures (like trees) on GPUs [7]. Our work does not replace these existing approaches, but instead, complements them by allowing the optimizations to be generated automatically.

7 Conclusion

We proposed techniques for efficient GPU code generation of graph algorithms from their high-level description. We reused an existing graph analytics DSL, Green-Marl, as the front-end and added a CUDA backend called `LightHouse`. It had to overcome several challenges specific to GPUs due to separate memories, thread-hierarchy and SIMD processing on GPUs. We discussed unique issues encountered in GPU code generation compared to those in CPU code generation. We illustrated the effectiveness of our approach by generating CUDA code for four graph algorithms and comparing their performance against that of their OpenMP versions generated by Green-Marl. The performance benefits reveal that DSLs provide an effective way of developing parallel algorithms.

References

1. D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. ICPP'06, pages 523–530.
2. A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. SC '11, pages 65:1–65:12. ACM, 2011.
3. M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IISWC '12*, pages 141–151. IEEE Computer Society, 2012.
4. F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. SC '12, pages 13:1–13:12.

5. A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *PACT'12*, 2012.
6. S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS 2012*, pages 349–362. ACM, 2012.
7. T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *CGO'12*. ACM, 2012.
8. M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP 2009*, pages 3–14, 2009.
9. M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008.
10. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI*, 42(6):211–222, 2007.
11. J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.
12. K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.
13. R. Nasre, M. Burtscher, and K. Pingali. Morph Algorithms on GPUs. In *PPoPP '13*, PPoPP '13. ACM, 2013.
14. R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. SC '10, pages 1–11.
15. 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 *PLDI'11*, pages 12–25. ACM, 2011.
16. D. Prountzos, R. Manevich, and K. Pingali. Elixir: a system for synthesizing concurrent graph programs. In *OOPSLA '12*, pages 375–394. ACM, 2012.
17. D. Prountzos, R. Manevich, and K. Pingali. Synthesizing Parallel Graph Programs via Automated Planning. In *PLDI 2015*, pages 533–544. ACM, 2015.
18. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI '13*, pages 519–530. ACM, 2013.
19. J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP 2013*, pages 135–146. ACM, 2013.
20. A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.
21. S. Xiao and W. chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.
22. A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *ICS 2005*, pages 25–. IEEE Computer Society, 2005.
23. E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS 2011*. ACM, 2011.
24. J. Zhong and B. He. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, 2014.