A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System

Prasanth Chatarasi and Vivek Sarkar
Georgia Institute of Technology
Atlanta, Georgia, USA
{cprasanth,vsarkar}@gatech.edu

ABSTRACT

Unlike dense linear algebra applications, graph applications typically suffer from poor performance because of 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Hence, there is a rapid growth in improving both software and hardware platforms to address the above challenges. One such improvement in the hardware platform is a realization of the Emu system, a thread migratory and near-memory processor. In the Emu system, a thread responsible for computation on a datum is automatically migrated over to a node where the data resides without any intervention from the programmer. The idea of thread migrations is very well suited to graph applications as memory accesses of the applications are irregular. However, thread migrations can hurt the performance of graph applications if overhead from the migrations dominates benefits achieved through the migrations.

In this preliminary study, we explore two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging hardware support for remote atomic updates to address overheads arising from thread migration, creation, synchronization, and atomic operations. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500.—Conductance, Bellman-Ford's algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MCHPC'18, November 11, 2018, Dallas, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6113-2/18/11…$15.00
https://doi.org/10.1145/3286475.3286481

KEYWORDS

Loop fusion, Edge flipping, Graph algorithms, Thread migratory, Near-memory, Atomic operations, The Emu system, Compilers

ACM Reference Format:
Prasanth Chatarasi and Vivek Sarkar. 2018. A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3286475.3286481

1 INTRODUCTION

Though graph applications are increasing in importance with the advent of "big data", achieving high performance with graph algorithms is non-trivial and requires careful attention from programmers [20]. Two significant bottlenecks to achieving higher performance on existing CPU and GPU-based architectures are 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Since graph applications are typically cache-unfriendly and are not well supported by existing traditional architectures, there is growing attention being paid by the architecture community to innovate suitable architectures for such applications. One such innovation is the Emu system, a highly scalable near memory system with support for migrating threads without programmer intervention [8]. The system is designed to improve the performance of data-intensive applications exhibiting weak locality, i.e., from irregular and cache-unfriendly memory access which are often found in graph analytics [18] and sparse matrix algebra operations [19].

Emu architecture. An Emu system consists of multiple Emu nodes interconnected by a fast rapid IO network, and each node (shown in Figure 1) contains *nodelets*, stationary cores and migration engines. Each *nodelet* consists of a Narrow Channel DRAM (NCDRAM) memory unit and multiple Gossamer cores, and the co-location of the memory unit with the cores makes the overall Emu system a near-memory system. Even though each nodelet has a different physical co-located memory unit, the Emu system provides a logical view

RIGHTSLINK

of the entire memory via the partitioned global address space (PGAS) model with memory contributed by each nodelet.

Each gossamer core of a nodelet is a general-purpose, simple pipelined processor with no support for data caches and branch prediction units, and is also capable of supporting 64 concurrent threads using fine-grain multi-threading. A key aspect of the Emu system is thread migration by hardware, i.e., a thread is migrated on a remote memory read by removing thread context from the nodelet and transmitting the thread context to a remote nodelet without programmer intervention. As a result, each nodelet requires multiple queues such as service, migration and run queues to process threads spawned locally (using *spawn* instruction) and also migrated threads.

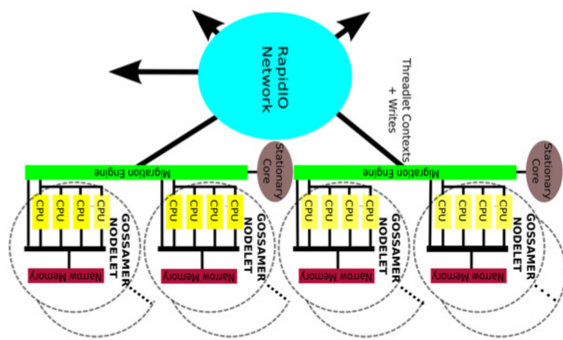

Figure 1: Overview of a single Emu node (Figure source: [10]), where a dotted circle represents a nodelet. Note that, the co-location of the narrow channel memory unit (NCDRAM) with gossamer cores makes the overall Emu system a near memory system.

Software support. The Emu system supports the Cilk parallel programming model for thread spawning and synchronization using `cilk_spawn`, `cilk_sync` and `cilk_for` constructs [11]. Since the Emu hardware automatically takes care of thread migration and management; hence the Cilk runtime is discarded in the toolchain. Also, it is important to note that appending a `cilk_spawn` keyword before a function invocation to launch a new task is directly translated to the `spawn` instruction of the Emu ISA during the compilation. The Emu system also provides libraries for data allocation and distribution over multiple nodelets, and intrinsic functions for atomic operations and migrating thread control functions. Also, there has been significant progress made in supporting standard C libraries on the Emu system.

Even though the Emu system is designed to improve the performance of data-sensitive workloads exhibiting weak-locality, the thread migrations across nodelets can hamper the performance if overhead from the thread migration dominates the benefits achieved through the migration. In the next section, we study both high-level and low-level compiler transformations which can be applied to original graph applications to mitigate the overheads as mentioned earlier.

2 COMPILER TRANSFORMATIONS

In this section, we discuss two high-level compiler transformations (*Node fusion* and *Edge flipping*)[1], and one low-level compiler transformation leveraging the *remote atomic update* feature of the hardware, to mitigate the impact of overheads in the performance of graph applications on the Emu system.

2.1 Node/Loop Fusion

Programmers write graph applications with multiple parallel loops over nodes of a graph either to 1) compute various properties of a node (e.g., in *Conductance* [7, 21]), or 2) query on computed properties of nodes (e.g., in *Average teenage followers* [17]). In such scenarios, multiple such parallel loops can be grouped into a single parallel loop, and compute multiple properties in the same loop or query immediately after computing the properties. This grouping can result in reducing thread migrations occurring in later loops, and also overheads arising from thread creation and synchronization. The grouping of multiple such parallel loops is akin to loop fusion, a classical compiler transformation for improving locality; but we use the transformation to reduce unnecessary migrations (for more details, see Section 3.2).

2.2 Edge Flipping

Edge flipping is another compiler transformation discussed in [15] to flip a loop over incoming edges of a node with outgoing edges of the node. However, we generalize the edge flipping transformation to allow flips between both incoming and outgoing edges. To allow this bi-directional flipping, the transformation assumes an input graph to be bi-directional, i.e., each node in the graph stores a list of incoming edges along with outgoing edges.

Vertex centric graph algorithms such as Page rank, Bellman-Ford algorithm for single-source shortest path, Page coloring offer opportunities to explore the edge flipping transformation since these algorithms either explore incoming edges of a node to avoid synchronization (pull-based approach), or explore outgoing edges to reduce random memory accesses (push-based approach), or explore a combination [6, 29]. We discuss the above push-pull dichotomy in Section 2.2, using

[1]Note that these high-level transformations – node fusion and edge flipping – have already been explored in past work on optimizing graph algorithms on the x86 architectures [15], and we are evaluating them in the context of the EMU system in this paper.

RIGHTSLINK

the Bellman-Ford's algorithm as a representative of vertex-centric graph algorithms.

2.3 Use of Remote Updates

Remote updates, one of the architectural features of the Emu, are stores and atomic operations to a remote memory location that don't require returning a value to the thread, and these operations do not result in thread migrations [8]; instead they send an information packet to the remote nodelet containing the data and the operation to be performed. These remote updates also can be viewed as very efficient special-purpose migrating threads, and they don't return a result unlike regular atomic operations, but they return an acknowledgement that the memory unit of the remote nodelet has accepted the operation. We leverage this feature as a low-level compiler transformation replacing regular atomic operations that don't require returning a value by the corresponding remote updates. The benefits of this transformation can be immediately seen in vertex-centric algorithms (Section 3.3) and also in the triangular counting algorithm (Section 3.4).

3 EXPERIMENTS

In this section, we present the benefits of applying the compiler transformations on graph algorithms. We begin with an overview of the experimental setup and the graph algorithms used in the evaluation, and then we present our discussion on preliminary results for each algorithm.

3.1 Experimental Setup

Our evaluation uses dedicated access to a single node of the Emu system, i.e., the Emu Chick prototype[2] which uses an Arria 10 FPGA to implement Gossamer cores, migration engines, and stationary cores of each nodelet. Table 1 lists the hardware specifications of a single node of the Emu Chick.

Table 1: Specifications of a single node of the Emu system.

	Emu system
Microarch	Emu1 Chick
Clock speed	150 MHz
#Nodelets	8
#Cores/Nodelet	1
#Threads/Core	64
Memorysize/Nodelet	8 GB
NCDRAM speed	1600MHz
Compiler toolchain	emusim.HW.x (18.08.1)

In the following experiments, we compare two experimental variants: 1) Original version of a graph algorithm running with all cores of a single node and 2) Transformed version after manually applying compiler transformations on the graph algorithm. In all experiments, we measure only the execution time of the kernel and report the geometric mean execution time measured over 50 runs repeated in the same environment for each data point. The speedup is defined as the execution time of the original version of a graph algorithm divided by the execution time of the transformed version of the program running with all cores of a single node of the Emu system in both cases,i.e., eight cores.

We also use an in-house simulation environment of the Emu prototype, whose specifications match with the hardware details mentioned in Table 1, to measure statistics of programs such as thread migrations, threads created and terminated. We are not currently aware of any methods for extracting these statistics from the hardware. We define the percentage reduction in thread migrations[3] as follows:

$$\%\text{reduction in migrations}$$

$$= \left(1 - \left(\frac{\#\text{migrations in the transformed version}}{\#\text{migrations in the original version}}\right)\right) \times 100$$

Finally, we evaluate the benefits of compiler transformations by measuring both improvements in execution time on the Emu hardware and reduction in thread migrations on the Emu simulator.

Graph applications: For our evaluation, we consider three graph algorithms, i.e., 1) Conductance algorithm, 2) Bellman-Ford's algorithm for Single-source shortest path (SSSP) problem, and 3) Triangle counting algorithm. Both original and transformed versions of above algorithms are implemented using the Meatbee framework [13], an in-house experimental streaming graph engine used to develop graph algorithms for the Emu system. The Meatbee framework, inspired by the STINGER framework [9], uses a striped array of pointers to distribute the vertex array across all nodelets in the system, and also implements the adjacency list as a hash table with a small number of buckets.

Input data-sets: We use RMAT graphs (edges of these graphs are generated randomly with a power-law distribution), scale[4] from 6 to 14 as specified by Graph500 [2]. Note that all the above graphs specified by Graph500 are generated using the utilities present in the STINGER framework. Table 2 presents details of the RMAT graphs used in our evaluation, and total thread migrations and execution times of the original graph algorithms on the Emu system.

[2]Several aspects of the system are scaled down in the prototype Emu system, e.g., number of gossamer cores of a nodelet

[3]Note that the thread migration counts are for the entire program, and we are not currently aware of any existing approaches on how to obtain migration counts for a specific region of code.

[4]A scale of n for an input graph refers to having 2^n vertices.

RIGHTSLINK

Scale	#vertices	#edges	Thread migrations in the original program			Execution time of the original program (ms), geometric mean of 50 runs		
			Conductance	SSSP-BF	Triangle counting	Conductance	SSSP-BF	Triangle counting
6	64	1K	6938	10915	26407	4.45	26.32	53.63
7	128	2K	13812	22851	84168	7.51	393.04	163.36
8	256	4K	28221	48354	252440	13.89	1634.64	547.84
9	512	8K	59068	104653	809423	32.13	2887.61	1694.09
10	1K	16K	122088	220204	2475350	64.59	4589.42	3942.55
11	2K	32K	253364	474118	7381977	134.43	10225.10	12649.30
12	4K	64K	522530	1136600	21777902	844.38	32140.30	36199.60
13	8K	128K	1065640	2332741	64063958	1841.53	-	185864.00
14	16K	256K	2171311	4569519	180988114	7876.99	-	721578.00

Table 2: Experimental evaluation of three graph algorithms (Conductance, SSSP-BF and Triangle counting) on the RMAT graphs from scales 6 to 14 specified by Graph500. Transformations applied on the algorithms: Conductance/SSSP-BF/Triangle counting: (Node fusion)/(Edge flipping and Remote updates)/ (Remote updates). The evaluation is done a single node of the Emu system described in Table 1. Note that we had intermittent termination issues while running SSSP-BF from scale 13-14 on the Emu node, and hence we omitted its results.

3.2 Conductance algorithm

The conductance algorithm is a graph metric application to evaluate a graph partition by counting the number of edges between nodes in a given partition and nodes in other graph partitions [7, 21]. The algorithm is frequently used to detect community structures in social graphs. An implementation of the conductance algorithm is shown in Algorithm 1. The implementation[5] at a high-level consists of three parallel loops iterating over vertices of a graph to compute different properties (such as `din, dout, dcross`) of a given partition (specified as `id` in the algorithm). Finally, these properties are used to compute conductance value of the partition of the graph.

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (v) of the graph from the first parallel loop (lines 2-4), and migrates to a nodelet where the vertex property `partition_id` is stored after encountering the property (`v.partition_id`) at line 3. Since the `degree` property of the vertex (v) is also stored on the same nodelet as of the other property[6], the thread doesn't migrate on encountering the property, `v.degree`, at line 4. After reduction of the `din` variable, the hardware performs a global synchronization of all spawned threads because of an implicit barrier after the parallel loop. After the synchronization, the hardware again spawns a thread for every vertex from the second parallel loop (lines 5-7), and migrates after encountering the same property (`v.partition_id` at line 6). The same behavior is repeated in the third parallel loop as well (lines 8-12). The repeated migrations to the same nodelet from multiple parallel loops, which arise from accessing the same property or a different property which is stored on the same nodelet, can be reduced by fusing all the three parallel loops into a single loop. Also, the fusion of multiple parallel loops can reduce the overhead of multiple thread creations and synchronization. As can be seen from Figure 2, we have observed a geometric mean reduction of 6.06% in the total number of thread migrations after fusing three loops. As a result, we also found a geometric mean speedup of 1.95x in the execution time of the computation over the scale 6-14 of RMAT graphs specified by Graph500. This performance improvement demonstrates the need for fusing parallel loops over nodes of a graph to compute values/properties together

Algorithm 1: An implementation of the Conductance algorithm [7, 21].

```
1  def CONDUCTANCE(V, id):
2  for each v ∈ V do in parallel with reduction
3      if v.partition_id == id then
                    ▷ Thread migration for v.partition_id value
4          din+ = v.degree
5  for each v ∈ V do in parallel with reduction
6      if v.partition_id != id then
7          dout+ = v.degree
8  for each v ∈ V do in parallel with reduction
9      if v.partition_id == id then
10         for each nbr ∈ v.nbrs do
11             if nbr.partition_id != id then
12                 dcross+ = 1
13 return dcross/((din < dout)?din : dout)
```

[5]The implementation is from a naive translation from existing graph analytics domain-specific compilers for non-EMU platforms.

[6]The properties of vertices (such as `partition_id`, `degree`) are allocated similar to the vertex allocation, i.e., uniformly across all nodelets.

RIGHTSLINK

to reduce thread migrations in applications such as Conductance.

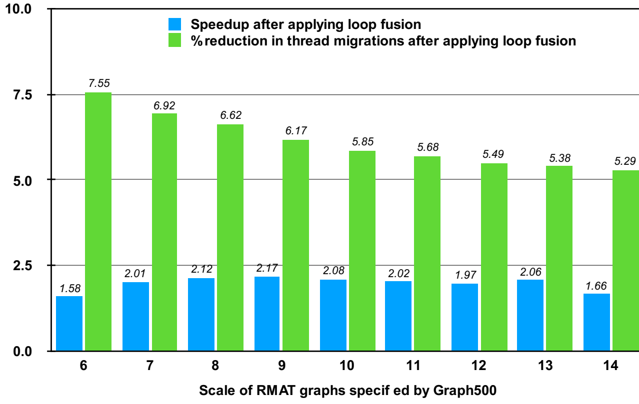

Figure 2: Speedup over the original conductance algorithm on a single Emu node (8 nodelets) and % reductions in thread migrations after applying loop fusion.

3.3 Single Source Shortest Path using Bellman-Ford's Algorithm (SSSP-BF)

Bellman-Ford's algorithm is used to compute shortest paths from a single source vertex to all the other vertices in a weighted directed graph. An implementation of the algorithm is shown in Algorithm 2. We added a minor step (at lines 15, 18, 23-25) in the body of the `t-loop` to the implementation for termination if subsequent iterations of the `t-loop` will not make any more changes, i.e., the distance computed (`temp_distance`) for each vertex in the current iteration is the same as the distance in the previous iteration (`distance`).

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (v) of the graph from the parallel loop (lines 6-13) nested inside the `t-loop`. The thread responsible for a particular vertex (v) in a given iteration (t) migrates to an incoming neighbor vertex (u) on encountering the accesses `distance(`u`)` and `weight(u, v)` (line 8). After adding the values, the thread migrates back to the original node for writing after encountering the access `temp_distance(u)` (line 9). The same migration behavior is repeated for every incoming neighbor vertex, and finally the local value based on the best distance from incoming neighbors is computed. This approach is commonly known as a pull-based approach since the vertex pulls information from incoming neighbors to update its local value. However, the back and forth migrations for every neighbor vertex via incoming edges can be avoided by doing the edge flipping transformation (discussed in Section 2.2), i.e., the loop iterating over incoming edges is flipped into a loop over outgoing

Algorithm 2: An implementation of the Bellman-Ford's algorithm (SSSP-BF).

```
1  def SSSP_BFS(V, id):
2  distance(id) ← 0
3  distance(v) ← MAX, for ∀v ∈ (V − {id})
4  temp_distance(v) ← 0, for ∀v ∈ V
5  for t ← 0 to |V| − 1 do
6      for each v ∈ V do in parallel
7          for each u ∈ incoming_neighbors(v) do
8              temp = distance(u) + weight(u, v)
                                ▹ Migration for distance(u) value
9              if distance(v) > temp then
10                 temp_distance(v) = temp
11             end
12         end
13     endfor
14
15     modified ← false
16     for each v ∈ V do in parallel
17         if distance(v)! = temp_distance(v) then
18             modified ← true
19             distance(v) = temp_distance(v)
20         end
21     endfor
22
23     if modified == false then
24         break;
25     end
26 end
27 return distance;
```

edges. The transformations leads to a push-based approach for the SSSP algorithm, in which a vertex pushes its contribution (`distance(u) + weight(u, v)`) to its neighbors accessible via outgoing edges and doesn't require migrating to the neighbors, as in the pull-based approach. Since multiple vertices can have a common neighbor, the contribution is done atomically, i.e., by using atomic_min in our implementation.

As a result of applying edge-flipping transformation, we have observed a geometric mean reduction of 8.69% in the total number of thread migrations (shown in Figure 3). However, the push-based approach with regular atomic updates didn't perform well compared with the pull-based approach from the scale of 7 to 9 (shown in Figure 4), because of irregularity in the input graphs and imbalance in the number of incoming and outgoing edges. As a result, the cost of migrating back and forth in the pull-based approach was not expensive compared to doing more atomic updates in the push-based approach for the above data points. This observation is in accordance with the push-pull dichotomy discussed in [6, 29].

RIGHTSLINK

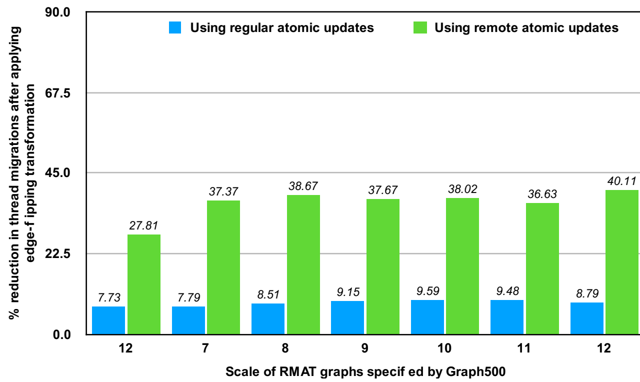

Figure 3: % reductions in thread migrations of SSSP-BF algorithm after applying edge flipping with regular atomic updates and with remote atomic updates on a single node (8 nodelets) of Emu Chick.

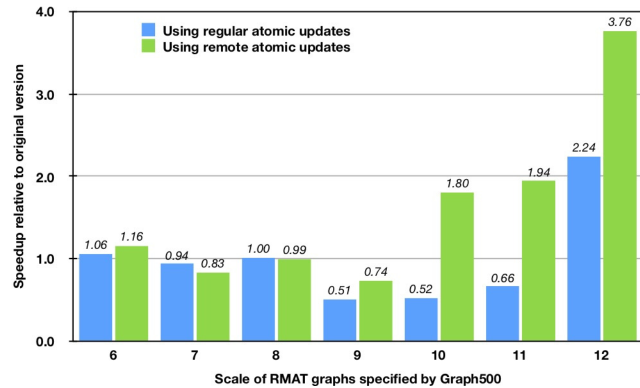

Figure 4: Speedup of SSSP-BF algorithm on a single Emu node (8 nodelets) after applying edge flipping with regular atomic updates and with remote updates.

Furthermore, the push-based approach can be strengthened by replacing regular atomic updates with remote atomic updates since a node which is pushing its contribution (i.e., its distance) to neighbors via outgoing edges doesn't need a return value. By doing so, we have observed a geometric mean reduction of 30.28% in thread migrations (shown in Figure 3) compared to the push-based approach with regular atomic updates. Also, there is an overall geometric mean improvement of 1.57x in execution time relative to the push-based approach with regular atomic updates (shown in Figure 4). The above performance improvement demonstrates the need for using remote atomic updates for scalable performance, and also exploring hybrid approaches involving both push and pull strategies based on input graph data.

3.4 Triangle Counting Algorithm

Triangle counting is another graph metric algorithm which computes the number of triangles in a given undirected graph, and also computes the number of triangles that each node belongs to [24]. The algorithm is frequently used in complex network analysis, random graph models, and also real-world applications such as spam detection. An implementation of the Triangle counting is shown in Algorithm 3, and it works by iterating over each vertex(v), picking two distinct neighbors (u, w), and check if there exists an edge between them to be part of a triangle. Also, the implementation avoids duplicate counting by delegating the counting of a triangle to the vertex with lower id.

Algorithm 3: An implementation of the Triangle counting algorithm [24].

1 $tc(v) \leftarrow 0$, for $\forall v \in (V)$
2 **for each** $v \in V$ **do in parallel**
3   **for each** $u \in v.nbrs$ **do**
4   **if** $nbr1 > v$ **then**
5   **for each** $w \in v.nbrs$ **do**
6   **if** $w > u$ **then**
7   **if** $edge_exists(u, w)$ **then**
8   tc_count ++; //Atomic
9   tc(v) ++; //Atomic
10   tc(u) ++; //Atomic
11   tc(w) ++; //Atomic
            ▷ Above regular atomics can be replaced by the remote updates.

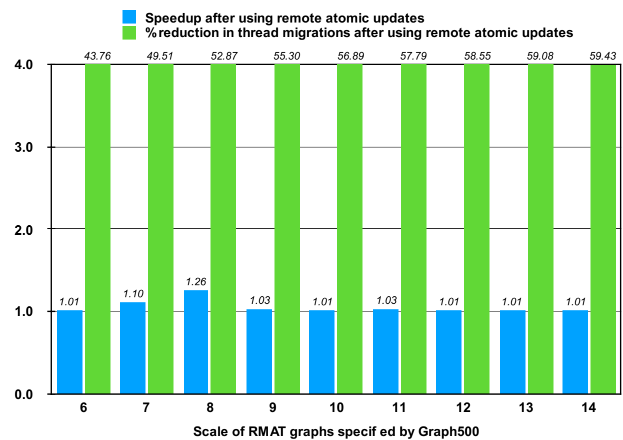

Figure 5: Speedup over the original triangle counting implementation on a single Emu node (8 nodelets) and % reductions in thread migrations after using remote atomic updates.

RIGHTSLINK Copyright Clearance Center

In the above implementation of the triangle counting algorithm, whenever a triangle is identified (line 7), the implementation atomically increments the overall triangles count and triangle counts of the three vertices of the triangle. As part of the atomic update operation, the thread performs a migration to the nodelet having the address. However, the thread incrementing the triangle counts doesn't need the return value of the increment for further computation; hence, the regular atomic updates can be replaced by remote atomic updates to reduce thread migrations. After replacing with remote updates, we have observed a geometric mean reduction of 54.55% in the total number of thread migrations (shown in Figure 5). As a result, we also found a geometric mean speedup of 1.05x[7] in the execution time of the kernel over the scale 6-14 of RMAT graphs specified by Graph500.

4 RELATED WORK

There is an extensive body of literature in optimizing graph applications for a variety of traditional architectures [3, 4, 27], accelerators [12, 16], and processing in memory (PIM) architectures [1, 22]. Also, there has been significant research done on optimizing task-parallel programs to reduce the overheads arising from task creation, synchronization [23, 25, 30] and migrations [26]. In this section, we discuss past work closely related to optimizing irregular applications for the Emu system and also past work on compiler optimizations in mitigating task (thread) creation, synchronization and migration overhead.

Emu related past work. Kogge et al. in [19] discussed migrating thread paradigm of the Emu system as an excellent match for systems with significant near-memory processing, and evaluated its advantage over a sparse matrix application (SpMV) and a streaming graph analytic benchmark (Firehose). Hein et al. [14] characterized the Emu chick hardware prototype (same as what we used in our evaluation) using custom kernels and discussed memory allocation, thread migrations strategies for SpMV kernels. In this work, we study high-level, and low-level compiler transformations that can benefit existing graph algorithms by leveraging the intricacies discussed in [5, 8, 14, 19, 28].

Programming models support and compiler optimizations for reducing thread creation, synchronization and migration overheads. Task-parallel programs often result in considerable overheads in task creation and synchronization, and hence approaches in [23, 25, 30] presented compiler frameworks to transform the input program to reduce the overheads using optimizations such as task fusion, task chunking, synchronization (`finish` construct) elimination. Our study on the loop fusion transformation to reduce thread creation and synchronization overheads on the Emu system is inspired by the above compiler frameworks and also from the Green-Marl DSL compiler [15].

[7]Note that the computational complexity of the triangle counting algorithm is significant, i.e., $O(m^{\frac{3}{2}})$ where m is number of edges, and even 5% improvement is equivalent to few thousands of msecs as reported in Table 2.

5 CONCLUSIONS AND FUTURE WORK

Graph applications are increasing in popularity with the advent of "big data", but achieving higher performance is not trivial. The major bottlenecks in graph applications are 1) inefficient utilization of memory subsystems through random memory accesses to the graph data, and 2) overhead of executing atomic operations. Since these graph applications are cache-unfriendly and are not well handled by existing traditional architectures, there is growing attention in the architecture community to innovate suitable architectures for such applications.

One of the innovative architecture to handle graph applications is a thread migratory architecture (Emu system), where a thread responsible for computation on a data is migrated over to a nodelet where the data resides. However, there are significant challenges which need to be addressed to gain the potential of Emu system, and they are reducing thread migration, creation, synchronization, and atomic operation overheads. In this study, we explored two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging remote atomic updates to address the above challenges. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500.—Conductance, Bellman-Ford's algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations. This preliminary study clear motivates us in exploring the implementation of automatic compiler transformations to alleviate the overheads arising from running graph applications on the Emu system.

6 ACKNOWLEDGMENTS

We would like to thank Eric Hein for his help in getting us started with the Emu system and using the Meatbee framework to develop graph algorithms. Also, we would like to acknowledge Jeffrey Young for his help with the Emu machine at Georgia Tech.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

RIGHTSLINK

[2] David A Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E Jason Riedy, and Jeremiah Willcock. 2011. *Graph500 Benchmark 1 (search) Version 1.2*. Technical Report. Graph500 Steering Committee.

[3] David A. Bader and Kamesh Madduri. 2005. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)*. Springer-Verlag, Berlin, Heidelberg, 465–476. https://doi.org/10.1007/11602569_48

[4] D. A. Bader and K. Madduri. 2006. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *2006 International Conference on Parallel Processing (ICPP'06)*. 523–530. https://doi.org/10.1109/ICPP.2006.34

[5] Mehmet E. Belviranli, Seyong Lee, and Jeffrey S. Vetter. 2018. Designing Algorithms for the EMU Migrating-threads-based Algorithms. In *High Performance Extreme Computing Conference (HPEC), 2018 IEEE*. IEEE.

[6] Maciej Besta, MichałPodstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/3078597.3078616

[7] Bela Bollobas. 1998. *Modern Graph Theory*. Springer.

[8] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2–9. https://doi.org/10.1109/IA3.2016.007

[9] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[10] EmuTechnology. 2017 (accessed December 12, 2017). *Emu System Level Architecture*. http://www.emutechnology.com/products/#lightbox/0/

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[12] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[13] Eric Hein. 2017. Meatbee, An Experimental Streaming Graph Engine. https://github.gatech.edu/ehein6/meatbee.

[14] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy. 2018. An Initial Characterization of the Emu Chick. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 579–588. https://doi.org/10.1109/IPDPSW.2018.00097

[15] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 349–362. https://doi.org/10.1145/2150976.2151013

[16] S. Hong, T. Oguntebi, and K. Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 78–88. https://doi.org/10.1109/PACT.2011.14

[17] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 208, 11 pages. https://doi.org/10.1145/2544137.2544162

[18] P. M. Kogge. 2017. Graph Analytics: Complexity, Scalability, and Architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1039–1047. https://doi.org/10.1109/IPDPSW.2017.176

[19] Peter M. Kogge and Shannon K. Kuntz. 2017. A Case for Migrating Execution for Irregular Applications. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms (IA3'17)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3149704.3149770

[20] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel Graph Analytics. *Commun. ACM* 59, 5 (April 2016), 78–87. https://doi.org/10.1145/2901919

[21] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1, Article 1 (July 2016), 20 pages. https://doi.org/10.1145/2898361

[22] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[23] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[24] Thomas Schank. 2007. *Algorithmic Aspects of Triangle-Based Network Analysis*. Ph.D. Dissertation. Universität Karlsruhe.

[25] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. 2009. Chunking Parallel Loops in the Presence of Synchronization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/1542275.1542304

[26] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer-Verlag, Berlin, Heidelberg, 172–187. https://doi.org/10.1007/978-3-642-13374-9_12

[27] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, Washington, DC, USA, 25–. https://doi.org/10.1109/SC.2005.4

[28] Jeffrey Young, Eric Hein, Srinivas Eswar, Patrick Lavin, Jiajia Li, Jason Riedy, Richard Vuduc, and Tom Conte. 2018. A Microbenchmark Characterization of the Emu Chick. arXiv:cs.DC/1809.07696

[29] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt - A High-Performance DSL for Graph Analytics. *CoRR* abs/1805.00923 (2018). arXiv:1805.00923 http://arxiv.org/abs/1805.00923

[30] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. 2010. Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs. In *Proceedings of the 19th International Conference on PACT*. ACM, New York, NY, USA, 169–180.

RIGHTSLINK