Linear Algebra-Based Triangle Counting via Fine-Grained Tasking on Heterogeneous Environments

(Update on Static Graph Challenge)

Abdurrahman Yaşar†, Sivasankaran Rajamanickam∗, Jonathan Berry∗, Michael Wolf∗, Jeffrey S. Young‡, and Umit V. Çatalyürek†
ayasar@gatech.edu, sr rajama@sandia.gov, jberry@sandia.gov, mmwolf@sandia.gov, jyoung9@gatech.edu, and umit@gatech.edu
∗Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, U.S.A.
†College of Computing, Georgia Institute of Technology, Atlanta, GA, U.S.A.

Abstract—Triangle counting is a representative graph problem that shows the challenges of improving graph algorithm performance using algorithmic techniques and adopting graph algorithms to new architectures. In this paper, we describe an update to the linear-algebraic formulation of the triangle counting problem. Our new approach relies on fine-grained tasking based on a tile layout. We adopt this task based algorithm to heterogeneous architectures (CPUs and GPUs) for up to 10.8x speed up over past year’s graph challenge submission. This implementation also results in the fastest kernel time known at time of publication for real-world graphs like twitter (3.7 second) and friendster (1.8 seconds) on GPU accelerators when the graph is GPU resident. This is a 1.7 and 1.2 time improvement over previous state-of-the-art triangle counting on GPUs. We also improved end-to-end execution time by overlapping computation and communication of the graph to the GPUs. In terms of end-to-end execution time, our implementation also achieves the fastest end-to-end times due to very low overhead costs.

I. INTRODUCTION

With increased use of accelerators for achieving better performance, proposing algorithms that require ideally no architecture specific changes on code base is crucial for portability on heterogeneous environments. This paper addresses these two primary problems, using general purpose accelerators for the graph challenge and doing that in a portable manner.

In this paper, we focus on a triangle counting algorithm that utilizes both CPUs and GPUs on a compute node and uses a portable tiled layout that requires almost no (algorithmic or layout) change between different architectures. This paper improves our previous work [1], [2] by using the linear algebra based triangle counting algorithm on a tiled layout and exploiting multiple levels of shared-memory parallelism on CPUs and GPUs.

This algorithmic changes allow us to achieve the fastest times for real world graphs like twitter and friendster compared to past champions utilizing the GPUs. We achieve 3.7 seconds on twitter and 1.8 seconds on friendster graphs as opposed to 6.5 and 2.1 seconds by past champions [3] when the graph is GPU resident. However, we believe assuming the graph is on the GPU is not realistic for several use cases, therefore in this paper we also focus on an end-to-end time metric when the graph is not GPU resident. In this case, we are able to count triangles in twitter and friendster graphs in 4.6 and 3.1 seconds where data copy is overlapped with computation. In the following sections we only report end to end results. In this paper, we propose a new linear algebraic formulation for triangle counting problem that uses tiles and a fine-grained parallel algorithm that exploits multiple level of parallelism on different architectures. We implement a highly efficient multi-core, multi-GPU hybrid framework that outperforms state-of-the-art. Experimental results demonstrate that our codes achieve the fastest kernel times on real-world graphs when the graph resides on the GPU and the fastest end-to-end times when the graph is not on the GPU. The performance improvement is up to 10× over our previous state-of-the-art implementation.

II. BACKGROUND

A. 2017 Static Graph Challenge

We used a linear algebra-based triangle counting implementation, KKTri (previously designated TCKK) [1] in the the 2017 Static Graph Challenge [4]. That work, focused on efficient shared memory parallelism on top of a portable SpGEMM (called KK-MEM) [5] in the Kokkos Kernels library [6]. The primary focus of that work was on two linear algebra based formulations of triangle counting:

1) \( D = (L \times U) \times L \): This formulation represented triangle counting in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication where \( L \) and \( U \) are the lower and upper triangle parts of the adjacency matrix for the graph.

2) \( D = (L \times L) \times L \): This formulation was used primarily for the 2017 Graph Challenge. This formulation follows the same logic as the previous method.

Three optimizations were used to achieve good performance: (1) in-place masked SpGEMM which reduced the memory needed for triangle counting; (2) data compression on the right hand side matrix that allowed using efficient bitwise operations (3) ordering of the vertices a common heuristic to reduce number of operations.

B. 2018 Static Graph Challenge

For the 2018 Static Graph Challenge [7], we designed a linear algebra-based triangle counting implementation KKTriCilk that inherited from the KK-SpGEMM algorithm [8] and
improved load-balancing and efficient hyper-thread usage issues using Cilk based programming model and optimizations.

The parallelization strategy and the runtime system is the main difference between KKTri-Cilk and KKTri. KKTri used a very simple scheme, partitioning the matrix evenly into partitions of a fixed number of rows. To balance the work among the tasks, KKTri-Cilk uses an heuristic to find the partitions, creating partitions such that the number of non-zeros within each partition are approximately equal.

Compression of the right hand side matrix can decrease the problem size significantly, and allow using efficient bitwise operations. However, compression is not always successful because of the natural order of the matrices.

III. APPROACH

A lightweight 2D partitioning algorithm is implemented to create tiles. This algorithm partitions the graph in two dimensional space where diagonal tiles are required to be squares. This partitioning is known as symmetric generalized block distribution [9].

In Equation 1, we propose a new linear algebra based triangle counting formulation that uses tiles. Similar to LL and LU this formulation represents triangle counting in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication but with tiles. If number of tiles is one then this formulation is identical with LL Algorithm [1]. Figure 1 illustrates this formulation. In the context of this paper, a “task”, t, counts the triangles in a given triple of tiles, \( t = \{T_{i,j}, T_{j,k}, T_{i,k}\} \). However, counting triangles in all possible triple of tiles, ends up counting each triangle three times. Considering tasks, \( t = \{T_{i,j}, T_{j,k}, T_{i,k}\} \) where \( i \leq j \leq k \) avoids unnecessary counting. With this restriction, if a given graph partitioned into \( p \times p \) tiles, then number of tasks, \( N_{task} \), is defined as \( N_{task} = \frac{p \times (p+1) \times (p+2)}{6} \). Tile based triangle counting can be formulated as:

\[
D_{i,j,k} = (T_{i,j} \times T_{j,k}) \ast T_{i,k} \tag{1}
\]

Latapy et al. [10] proposes to use list intersection based counting for small degree vertices and a hashmap based intersection for the other. In this work, a similar approach is used; i.e. any task with sparse tiles will use list based intersection and denser tiles will use a dense hashmap accumulator.

We use both CPUs and GPUs to process tasks together. There is no architecture specific algorithmic change in the code-base. This will allow execution on any accelerator and CPU combination. However, because of architectural differences, the parallelization approach differs between CPU and GPU itself. While, CPU threads execute different tasks in parallel, GPU threads execute cooperatively to complete the same task in parallel. Assigning heavier tasks to GPUs is one of the primary optimizations that is being used in hybrid approaches [11] due to massive parallelism capabilities of these devices. Hence, we try to estimate and execute heavier tasks on GPUs.

We use Cuda streams to simultaneously execute several tasks on the GPUs. Four Cuda streams are created for each GPU in the node. Then, a CPU thread is created and made responsible for the operations on the stream. GPUs and CPUs compete for tasks and get a new one from a queue when they are ready. Tasks are ordered based on their size in a task queue. GPUs start to process tasks starting from the heaviest task and CPUs start to process tasks from the lightest tasks. This continues until all tasks have been executed.

IV. EXPERIMENTAL EVALUATION

We present several experiments to identify the performance trade-offs of the proposed work. These experiments were carried out on three architectures with multicore processors and GPUs that are shown in Table I. GNU compiler (g++) version 7.2, Cuda runtime version 10.0 and OpenMP version 4.0 are used to compile and run the code on all the architectures.

A. Dataset and Copy Time Included Peak Rates

Table II lists 23 graphs that we used in our experiments along with the number of vertices (|V|), number of edges (|E|), number of triangles (|T|), size of the graph in memory (Raw Size), number of tiles and number of tasks in the graph. Note that when we partition the adjacency matrix into \( p \) intervals on rows and on columns in total \( p \times p \) tiles are created which is reported in Table II. However, our algorithm only uses upper triangular part of the matrix, therefore, only \( \frac{p \times (p+1)}{2} \) tiles are being used. In addition to 20 Challenge graphs for which triangle counting is particularly costly, 3 additional large real-world graphs [12], [13] are included in our experiments. We used the Graph Challenge procedure of symmetrizing the matrices (using undirected graphs). For all experiments, we
B. Relative Speedup comparisons to Last Year’s Submission

Figure 2 presents relative speedup between this work and our last submission (KKTri-Cilk) [2]. This work outperforms KKTri-Cilk in 20 of 23 cases. In three small instances (flickrEdges, amazon0505 and cit-Patents) KKTri-Cilk performs better. This years work can achieve up to $11 \times$ speedup on large graphs in which GPUs become more useful.

C. Effect of bandwidth on speedup

Figure 3 presents strong scaling of the friendster graph up to 4 GPUs on two machines; with NVLink and without NVLink. We observe from this figure that with NVLink enabled architecture we get better scaling. Hence, we believe that if we would have a server with 8 Volta GPUs with NVLink, we could get even better execution times than the reported ones.

V. CONCLUSIONS

We developed a fine-grained tasking based multi-core, multi-GPU, triangle counting method. This linear algebra implementation is up to $10.8 \times$ faster than our previous submission. This implementation results in the fastest end-to-end time known at time of publication due to very low overhead costs.

Acknowledgments: We thank Simon Hammond, Cynthia Phillips, and Stephen Olivier for helpful discussions. Sandia

REFERENCES


