Optimizing Approximate Weighted Matching on Nvidia Kepler K40

Md. Naim, Fredrik Manne*,
Mahantesh Halappanavar, Antonino Tumeo**
and Johannes Langguth***

Abstract

Matching is a fundamental graph problem with numerous applications in science and engineering. While algorithms for computing optimal matchings are difficult to parallelize, approximation algorithms on the other hand generally compute high quality solutions and are amenable to parallelization. In this paper, we present efficient implementations of the current best algorithm for half-approximate weighted matching, the Suitor algorithm, on Nvidia Kepler K-40 platform. We develop four variants of the algorithm that exploit hardware features to address key challenges for a GPU implementation. We also experiment with different combinations of work assigned to a warp. Using an exhaustive set of 269 inputs, we demonstrate that the new implementation outperforms the previous best GPU algorithm by 10 to 100× for over 100 instances, and from 100 to 1000× for 15 instances. We also demonstrate up to 20× speedup relative to 2 threads, and up to 5× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm. The new algorithms and implementations provided in this paper will have a direct impact on several applications that repeatedly use matching as a key compute kernel. Further, algorithm designs and insights provided in this paper will benefit other researchers implementing graph algorithms on modern GPU architectures.

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {md.naim, fredrikm}@ii.uib.no

**Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA.
Email: {mahantesh.halappanavar, antonino.tumeo}@pnnl.gov

***High performance computing, Simula Research Laboratory, Oslo, Norway. Email: langguth@simula.no

1 Introduction

Given a graph $G = (V, E)$ with vertex set V, edge set E and a weight function $w : E \rightarrow \mathbf{R}^+$, a matching M is a subset of edges such that no two edges in M are incident on the same vertex. A maximum matching maximizes the number of matched edges (cardinality) in M. The objective for a maximum weighted matching is to maximize the sum of the weights of the matched edges. Further, the solutions can be optimal or approximate. In this paper, we only consider half-approximate weighted matching algorithms that guarantee a solution that is at least half of an optimal solution in terms of the cardinality and weight of the matching. We present and compare two main algorithms, Locally-Dominant and Suitor, on two types of architectures, CPUs and GPUs. We also study four variants of the Suitor algorithm on GPUs. The algorithms are listed in Table 1, and are described in Sections 2 and 4.

Table 1: A list of matching algorithms and variants presented and studied in this paper.

Algorithm	**Description**
OMP-LD	Active vertices are stored in a shared queue. Presented in Algorithm 1.
OMP-Suitor	Uses locks for synchronization. Presented in Algorithm 3.
GPU-LD	Thread-per-vertex based implementation of OMP-LD.
GPU-Suitor	Warp-based implementation of OMP-Suitor. Synchronization and load balancing are not used. Presented in Algorithm 4.
GPU-Suitor-SyncLB	Synchronization and load balancing employed among participating warps.
GPU-Suitor-SyncNoLB	Synchronization, but no load balancing among warps.
GPU-Suitor-Hybrid	Synchronize and load balance only for the first few iterations of the Suitor algorithm.

Matching is a fundamental combinatorial problem with many applications in scientific computing, optimization and data analytics. In scientific computing, matchings are used in the solution of sparse linear systems to place large matrix elements on or close to the diagonal [4]; computation of sparse basis for the null space or column space of under-determined matrices [16]; computation of block-triangular form of a matrix [17]. Approximate weighted matchings are used in multi-level graph algorithms for partitioning and clustering during the coarsening

phase [10]; network alignment [11] and community detection [18]. These applications drive the need for efficient parallel implementations of matching algorithms on emerging multicore and manycore architectures. Many of these applications repeatedly compute matchings several hundreds of times during their execution. Therefore, small improvements in matching performance can lead to large gains in the performance of these applications [11].

An important class of manycore architectures are general purpose graphics programming units (GP-GPUs, or simply GPUs) that are not only powerful but also ubiquitous. The Nvidia Kepler K40 presented in Section 3 is currently one of the best manycore platforms for scientific computing. While many significant performance gains for compute intensive applications with regular and predictable memory access patterns have been demonstrated using GPUs, the efficient implementation of irregular applications such as graph algorithms remains a challenge [21]. Highly irregular degree distributions, poor locality in memory accesses, and minimal computation on accessed data make efficient utilization of compute resources challenging. Using approximate weighted matching as a case study for irregular applications, we introduce several algorithmic ideas that can also be adapted for other graph algorithms.

1.1 Contributions

We make the following contributions in this paper:

- Develop new parallel implementations of a weighted matching algorithm (GPU-Suitor) on the Nvidia Kepler architecture. We present four variants of the algorithm and several combinations of threads-per-block and vertices-per-warp.
- Present detailed experimental results using 269 test cases representing diverse applications and sparsity patterns (graph structures).
- Demonstrate the superiority of our algorithms over the previous best algorithm (GPU-LD) on GPUs [7], as well as shared-memory (OpenMP) implementations. We show that the new implementation outperforms GPU-LD by **10** to **100**× for over 100 instances, and by **100** to **1000**× for 15 instances. We also demonstrate up to **20**× speedup relative to 2 threads, and up to **20**× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm.

We organize the presentation in this paper as follows. We first present multithreaded matching algorithms targeting shared-memory architectures in Section 2. The Nvidia Kepler K40 is introduced in Section 3, followed by a discussion on the key challenges and our approaches to overcome them. GPU-Suitor, along with its four variants, are presented in Section 4. Experimental results and

analysis are presented in Section 5, followed by a discussion of related work in Section 6, and our conclusions in Section 7.

2 Parallel Weighted Matching

Matching is a classical topic in combinatorial optimization and has been studied extensively [12, 19, 15]. While many variants of the problem exist, we focus on approximate weighted matching for general graphs. In particular, we focus on the work of Halappanavar *et al.* on the Locally-Dominant Algorithm [7], and the work of Manne and Halappanavar on the Suitor Algorithm [13]. Their work itself was built on the pioneering work of many other researchers in the area, whose parallel algorithms systematically evolved from efficient serial algorithms. Due to space restrictions, we only present these two approximation algorithms in this section. We refer you to the respective papers for details.

2.1 The Locally-Dominant (LD) Algorithm

A half-approx weighted matching can be simply computed by considering the edges in a non-increasing order of weights, and by adding all edges that do not violate the matching condition. However, such an approach imposes a serial order on execution. Therefore, the main idea of the LD Algorithm is to identify and match *locally-dominant* edges in parallel. An edge that is heavier than all the edges incident on its end points is called a locally-dominant edge. Algorithm 1 implements this approach. It takes a graph $G = (V, E)$ as input and returns a matching M as output. The algorithm starts by making a call to Procedure ProcessVertex(v) for each vertex (Lines 6 and 7). In Procedure ProcessVertex, for a given vertex, all its neighbors are scanned to find the current heaviest neighbor that has not been matched already. It is important to break ties (duplicate weight) consistently to prevent deadlocks. For this purpose we use vertex indices, which are guaranteed to be unique (Line 5). The identity of the heaviest neighbor for each vertex is then stored in a vector (candidate). After setting the candidate mate for vertex s, say to vertex t, we check if the candidate mate for t is also set to s: candidate[candidate[s]] $= s$ (Line 9). If this is true, we have found a *locally-dominant* edge $e_{s,t}$. We add this edge to M, and the two vertices s and t to the queue (Line 12). Some of the vertices might end up not having any candidates available to match with.

The second part of the execution begins when every vertex has been processed and matched vertices have been added to the queue Q_C. In this part, we iterate until the queue becomes empty (Line 8 in Algorithm 1). Note that at least one edge (the heaviest edge) would get matched in the first loop, and therefore, Q_C is nonempty if M is nonempty. During each iteration of the **while** loop on Line 8,

Algorithm 1 Parallel Locally-Dominant Algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a queue, Q_C, listing vertices for processing in current step, and a queue, Q_N, listing vertices to be processed in the next step – both the queues list matched vertices; a vector candidate of size $|V|$ that contains the id of the current heaviest neighbor of each vertex.

```
1: procedure LOCALLY-DOMINANT(G(V, E), mate)
2:     for each v ∈ V in parallel do
3:         mate[v] ← ∅
4:         candidate[v] ← ∅
5:     Q_C ← ∅; Q_N ← ∅
6:     for each v ∈ V in parallel do
7:         PROCESSVERTEX(v, Q_C)
8:     while Q_C ≠ ∅ do
9:         for each u ∈ Q_C in parallel do
10:            for each v ∈ adj(u) do
11:                if candidate[v] = u then
12:                    PROCESSVERTEX(v, Q_N)
13:        SWAP(Q_C, Q_N)                          ▷ Swap the two queues
```

Algorithm 2 ProcessVertex

```
1: procedure PROCESSVERTEX(s, Q)
2:     max_wt ← −∞
3:     max_wt_id ← ∅
4:     for each t ∈ adj(s) do
5:         if (mate[t] = ∅) AND (max_wt < w(e_{s,t})) then
6:             max_wt ← w(e_{s,t})
7:             max_wt_id ← t
8:     candidate[s] ← max_wt_id
9:     if candidate[candidate[s]] = s then
10:        mate[s] ← candidate[s]
11:        mate[candidate[s]] ← s
12:        Q ← Q ∪ {s, candidate[s]}
```

we process vertices matched in the previous iterations while adding new vertices to the queue Q_N that become eligible as edges get matched. Note that we only need to process vertices for which the candidate was set to one of the matched vertices (Line 12). This is achieved by adding the newly matched vertices to the queue and checking if any of their unmatched neighbors point to them. If so, those neighbors will have to find new candidates for matching. The algorithm will terminate when the queue becomes empty. The matching is stored in a vector, mate.

The running time of Algorithm 1 is given by $O(|V| + |E|\Delta)$, where Δ is the

maximum degree in G. The worst case happens when a vertex points to all of its neighbors unsuccessfully, and in order to determine the current heaviest neighbor it needs to check the entire list. However, the runtime can be improved to $\Theta(|V|+|E|)$ if the adjacency list for each vertex is provided in a non-increasing order of edge weights. Under this assumption, the current heaviest neighbor of a vertex can be computed in constant time. The amount of parallelism is determined by the number of vertices in Q_C during each iteration of the **while** loop (Line 8). We use the compressed row storage format (CSR) for storing graphs in memory and therefore benefit from caching effects on adjacency lists on platforms with cache hierarchies. On the x86 platforms we use an intrinsic atomic operation `__sync_fetch_and_add()` to add vertices to the tail of the queue.

2.2 The Suitor Algorithm

We now present the Suitor algorithm, the currently best performing half-approx algorithm for weighted matching [13]. An important distinction of the Suitor algorithm relative to the Locally-Dominant algorithm is the absence of a central queue for active vertices that need to be considered for matching in a given iteration. Elimination of the queue makes the algorithm better suited for parallel implementation. Further, by paying careful attention to the vertex that is being processed, the Suitor algorithm proactively avoids unnecessary work. Similar to the Locally-Dominant algorithm, we again use vertex identities to break ties consistently. We also use the notion of locally-dominant edges in order to find candidate edges for matching. The Suitor algorithm is detailed in Algorithm 3.

Parallelism is achieved by distributing the executions of the outer **for** loop (Line 6 in Algorithm 3) among the threads. Multiple threads will concurrently process different vertices, and attempt to find a suitable candidate for each. Since two variables, mate and ws, are shared among the participating threads, there is a need for explicit synchronization among the threads. We use OpenMP locks for synchronization. To prevent conflicts, we define a lock for each vertex (Line 5) and then require that a thread must acquire a *partner*'s lock before executing lines 20 through 29, at which point the lock is released. Immediately after acquiring the lock we test if $heaviest > \mathsf{ws}[partner]$ is still true as it is possible that some other thread might have increased the value of $\mathsf{ws}[partner]$ after *partner* was determined to be the best match for *current*. If this is not the case, then *current* cannot be the suitor of *partner* and we must continue the search for next best candidate (lines 26 to 28). If a given vertex v ends up replacing another vertex w as the mate, then the thread processing vertex v becomes responsible for finding a suitable mate for w. This is shown in lines 21 to 23. The algorithm terminates when all the vertices have been processed. We note that there is no strict order in which the vertices need to be processed.

The running time of the serial Suitor algorithm is $O(\Sigma_{u\in V}|adj(u)|^2) = O(|E|\Delta)$

Algorithm 3 Parallel Suitor algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a vector ws of size $|V|$ that stores the weight of the current heaviest neighbor of each vertex.

```
1: procedure OMP-SUITOR(G(V,E), mate)
2:     for each u ∈ V in parallel do
3:         mate[u] ← NULL
4:         ws[u] ← 0
5:         omp_init_lock[u]                          ▷ Initialize the lock for each vertex
6:     for each u ∈ V in parallel do
7:         current ← u
8:         done ← False
9:         while (done = False) do
10:            partner ← mate[current]
11:            heaviest ← ws[current]
12:            next ← ∅
13:            for each v ∈ adj(current) do                ▷ For all neighbors of current
14:                if w(current, v) > heaviest and w(current, v) > ws(v) then
15:                    partner ← v
16:                    heaviest ← w(current, v)            ▷ Weight of edge (current, v)
17:            done ← True
18:            if heaviest ≠ NULL then ▷ True only if there is a candidate to match
                   with
19:                omp_set_lock[partner]                          ▷ Lock the partner
20:                if heaviest > ws[partner] then
21:                    if mate[partner] ≠ NULL then        ▷ Check if partner had a
                           previous offer
22:                        next ← mate[partner]
23:                        done ← False
24:                    mate[partner] ← current
25:                    ws[partner] ← heaviest
26:                else
27:                    done ← False                ▷ The partner already has a better offer
28:                    next ← u
29:                omp_unset_lock[partner]                ▷ Release the lock for partner
30:            if done = False then
31:                current ← next          ▷ Continue the search for next best candidate
```

as a node u might have to traverse its neighbor list $|adj(u)|$ times to find a new partner. Note that Δ is the maximum degree in G.

3 Architecture and Challenges

The NVIDIA Tesla K40, based on the Kepler architecture, is currently the most powerful single chip GPU board for scientific computing. There exists a dual chip board, Tesla K80, which trades off some of the peak performance of each chip to obtain higher combined performance and has better compute-to-shared-memory and register ratios, but requires multi-gpu programming techniques for effective utilization. The new GPUs based on the Maxwell architecture are more power efficient, but they are primarily targeted for single-precision computation and gaming applications. Their double precision performance is $\frac{1}{32}$ of the single precision performance.

The Tesla K40 features the GK110B GPU with 15 streaming multiprocessors (SMX). Each SMX integrates 192 single precision units, 64 double precision units and 32 special function units. Each SMX is equipped with 48 KB of read-only cache, as well as 64 KB of on-chip storage, configurable in splits of 48/16, 32/32 and 16/48 KB between L1 cache or shared-memory. The shared-memory is a directly addressable scratchpad memory. K40 also includes 1.5 MB of L2 cache shared among the 15 SMXes. The board provides 12 GB of GDDR5 memory with a datarate of 6 GHz, and all accesses to this memory are cached in L2 automatically. With a core clock of 745 MHz (and turbo clocks up to 875 MHz), the K40 has a theoretical peak performance of 4.29 TFLOPS (5 with turbo) in single-precision and 1.43 TFLOPS (1.66 with turbo) in double precision. Its peak memory bandwidth is 288 GB/s. Applications can reach about 80% of the peak bandwidth on the Kepler architecture [5]. Thus, in throughput oriented computing, it is significantly more powerful than current CPUs both for bandwidth- and compute-bound problems. However, its memory is limited to 12 GB, and even if K40 employs PCI-E v. 3.0 with bandwidths up to 16 GB/s, transfer rates between host and GPU memory are still an order-of-magnitude smaller than those between the GPU and its memory.

The GPU uses the Single Instruction Multiple Thread (SIMT) model, where threads are issued in warps (groups of 32 threads). A warp executes the same instruction at the same time for all its threads. Warps are further grouped into thread-blocks, which are sets of threads scheduled on the same SMX that can share data through the shared-memory. Finally, thread-blocks are organized in a grid, which comprises all threads launched in an application kernel. Since this is an important parameter, we provide results using several values of threads-per-block in Section 5.3.

3.1 Challenges in parallelization

The main challenges that limit performance on current GPU architectures are: (*i*) un-coalesced memory accesses, (*ii*) thread divergence, and (*iii*) load imbalance

among participating threads. The Kepler architecture somewhat mitigates the performance problems of un-coalesced memory accesses due to a better cache architecture. Memory accesses from the same warp that use the read-only cache can obtain maximum memory bandwidth independent of the thread ordering. Furthermore, shared memory can be used to coalesce other memory accesses.

Threads in the same warp are considered divergent if they take different paths in a branching statement. Because of their lockstep execution, all threads in a warp have to wait until threads that have taken different directions completes. Load imbalance among threads keeps warps executing longer on an SMX, thus wasting resources if only a handful of threads are still computing.

Therefore, efficient implementations on GPUs should address all of these challenges in a systematic manner. With reference to the previous implementation of approximate matching, we address these challenges by implementing the algorithm from the perspective of a warp processing a set of vertices instead of one thread processing a set of vertices. We will now briefly explain how we address the challenges in order to build towards the detailed presentation in Section 4.

Un-coalesced accesses in the previous (Locally-Dominant) implementation resulted from a thread-based approach where threads in a warp accessed neighborhoods of different vertices simultaneously, leading to poor locality of memory accesses and under-utilization of data caches. In our implementation, coalesced memory accesses are achieved by exploring the neighborhood of a vertex in parallel using the threads in a warp.

The performance of the previous implementation was also adversely impacted by thread divergence resulting from variations in the size of neighborhoods and vertex-specific decisions. Using the warp-based approach we minimize the impact of thread divergence. In our implementation, thread divergence is caused by threads in a warp attempting to set the suitor for the vertices that they are responsible for. For example, the availability of locks associated with the candidate-vertices that are chosen for a set of vertices in the warp, the branching of if statements based on the weights, and the replacement of one vertex by another vertex that needs to be processed further. While thread divergence is hard to eliminate, we tackle this challenge by exploring several combinations of vertices-per-warp. As presented in Section 5.3, best performance is observed with 8 vertices-per-warp. We note here that all the threads in a warp process the neighborhood of a vertex in tandem. Each vertex in a warp is processed in a sequential order. This is described in Section 4.

Variations in the sizes of the neighborhood (vertex degree) is a major source of load imbalance for approximate matching. This issue was not addressed in the previous implementation. For example, the slowest thread in a warp determined the speed of the warp. However, by using the warp-based approach to process the neighborhood of a vertex, load imbalance from varying vertex-degrees is minimized. We further address load imbalance by redistributing work among

participating warps of a thread block. The impact of this approach is presented in Section 5.3.

4 Weighted Matching on the GPUs

We now present the GPU implementations of the Suitor Algorithm. We build on the presentation of Suitor in Section 2. The GPU implementation of the Locally-Dominant Algorithm is a straight-forward adaptation of the multithreaded (OpenMP) algorithm, where a single thread processes the entire neighborhood of a vertex. In contrast, the GPU implementation of the Suitor Algorithm utilizes all the threads of a warp to processes the neighborhood of a vertex. Consequently, the vertices themselves are processed in serial on a given warp. In this section, we only present the GPU implementation of the Suitor Algorithm. We refer to Halappanavar *et al.* for details on the GPU implementation of the Locally-Dominant Algorithm [7].

The GPU implementation of the Suitor Algorithm utilizes the nested parallel structure of a GPU – where several warps run in parallel, and in turn each warp consists of parallel threads. Consequently, the fundamental difference between the OpenMP and GPU implementations arise from this nested structure. As an illustration, observe that in Algorithm 3 vertices are processed in parallel (Line 6). In contrast, chunks of vertices are assigned to concurrent warps (Line 2) for parallel execution in Algorithm 4. Each warp processes these vertices in serial (Line 5), but the neighborhood of a vertex is processed in parallel (Line 6). In the following discussion, we present intuition and details of the GPU adaptation of the Suitor Algorithm designed to maximize the nested parallelism of a GPU.

Algorithm 4 GPU-Suitor Algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Variables*: V_i represents a chunk of vertices based on vertices-per-warp processed on warp i.

```
1:  procedure GPU-Suitor(G(V, E), mate)
2:      Determine the number of warps required based on |V|, vertices-per-warp and
            threads-per-block
3:      while (there are vertices to process) do
4:          for each V_i in parallel do                          ▷ Across warps
5:              for each v ∈ V_i do
6:                  Process adj(v) in parallel                   ▷ In a warp
7:                  Determine best candidate for v in parallel
8:              Set suitor for each candidate of V_i in parallel
9:              Store self or displaced vertices                 ▷ Within a warp
10:             Synchronize across warps; load balance (optional)
```

We present the overall structure of the GPU-Suitor in Algorithm 4. The details are provided in the following discussion, where we also present the intuition

and differences among the four variants of GPU-Suitor. For ease of presentation, we present the algorithm in two phases: (*i*) Initial phase, and (*ii*) Recurrent phase.

Initial Phase

The algorithm starts by moving vertex indices from global memory of the GPU to the local (logical) shared-memory of each warp for a given chunk of vertices assigned to that warp (Line 2). Once a warp has read the indices of the neighbor lists for all vertices of its chunk into shared memory, it finds the best available candidate for each vertex v in the chunk consecutively. All the 32 threads in a warp collectively read the neighbor list of a vertex v (Line 6), and decide the best candidate using a butterfly reduction on the local best read by each thread in the warp (Line 7). As a result of this reduction, each thread of the warp discovers the best candidate and the weight of the corresponding edge. These values are saved in an intermediate buffer in the global memory or registers. After finding and storing all the candidates, the entire warp reads the intermediate buffer containing the stored candidates and corresponding edge weights in a coalesced manner. Each member thread is responsible for setting one vertex as the suitor of its corresponding candidate (Line 8). As detailed in Algorithm 3, a thread in a warp succeeds in setting its vertex as the mate of its candidate vertex if it has a heavier edge (ties resolved consistently). Similar to OMP-Suitor, locks are used in determining the current highest offer for a candidate stored in $\mathsf{ws}[partner]$.

If a thread fails to set a particular vertex v as the suitor of its best candidate c, or it succeeds in replacing a previously assigned vertex u, then we consider those vertices as unsuccessful. The threads of a warp collectively gather all the unsuccessful vertices in consecutive location of shared memory using parallel prefix sum. This is the same part of the memory that was initially used for storing vertex indices assigned to that warp.

The number of vertices assigned to a warp plays a critical role in determining the overall performance. We therefore use different values for vertices-per-warp and show the impact on performance in Section 5.3.

Recurrent Phase

Once a warp completes processing all the vertices in its chunk, it knows how many vertices need to be processed next (Line 9). Processing of these vertices can potentially lead to other vertices becoming eligible for processing in the next iteration. The warp keeps iterating over the recurrent phase until all of its vertices either obtain suitors or cannot be matched (no candidates are available). It is important to note that while each member thread in a warp can try for a different vertex in parallel, only one thread is allowed to set the suitor of the same vertex

at the same time. In order to avoid race conditions arising from this, we use atomic memory operations.

Synchronization and Load Balancing

Synchronization between warps can be avoided in a warp-based implementation, which can lead to minimization of the idle time of the multiprocessors. However, this can lead to an imbalanced load distribution among the warps of a block. For many inputs, we observed that most of the warps finish after a few iterations in the recurrent phase, while a few warps perform a significant number of iterations. To examine these effects further, we implemented intra-block load distribution among participating warps of a thread block. This distribution incurs a cost from synchronization between the warps of a block, and movement of unsuccessful vertices from the shared memory (logical) of one warp to the shared memory of other warp(s). Finding imbalances in load further requires atomic operations and logarithmic to linear operations to find warps with load deficiencies.

Depending on synchronization and load balancing, we have four variations of GPU-Suitor as described below. The differences in performance and their analysis is provided in Section 5.3.

1. **NoSync**: A thread block is not synchronized at all. Each warp works independently to match all of its vertices. This approach has both advantages and limitations. While most of the warps complete their work after a few iterations, only a few warps require tens to hundreds of iterations to complete. These numbers determine the overall performance of the kernel. Thus, load imbalance among the warps of a thread block has a large impact on overall performance. For inputs where most of the warps have approximately the same amount of work, the NoSync approach benefits immensely from avoiding thread synchronization within the blocks, which is an expensive operation on the GPU.

2. **SyncLB**: To alleviate problems arising from load imbalance in NoSync, this approach redistributes load among warps of a thread block during the first k iterations of the recurrent phase, which entails synchronization of all warps and thus of all threads. Here, k is either a predefined number or it is determined based on the number of vertices that haven't been the suitor for other vertices yet. As a result, all warps of a particular thread block are guaranteed to perform more or less the same work for these iterations. For subsequent iterations, warps of a block are synchronized in order to decide on termination of the block without any redistribution of the load. For our implementation, we allow a 25% deviation from the average load when redistributing work during the first k iterations.

3. **SyncNoLB**: In order to examine the impact of synchronization on execution time, this implementation synchronizes warps of a particular block in each iteration, but without balancing the load among them.

4. **Hybrid**: In preliminary experiments, we noticed that after first few iterations with load distribution, subsequent iterations takes more time with synchronization and load distribution than without any synchronization or load balancing. This variant performs load balancing only during the early phases of the execution.

5 Experimental Results and Analysis

We provide the experimental results and analysis in this section. In particular, we demonstrate significant speedup of the new algorithm and implementations relative to the previous best algorithm. We also demonstrate the speedup of GPU implementations relative to CPU (OpenMP) implementations. We further provide results on performance differences between different variants of the GPU implementation. Since we summarize the information in this section, we make the entire result set available at this website: `http://hpc.pnl.gov/people/hala/suitor.html`. The source code is available upon request.

5.1 Hardware Platforms and Dataset

All the experiments are conducted on a server with Intel CPUs and NVIDIA GPUs. The system integrates two sockets and 64 GB of DDR3-1600 memory. Each socket is equipped with an hyperthreaded 8-core Intel Xeon E5-2687W (Sandy Bridge) running at 3.10 GHz (turbo up to 3.8 GHz), thus amounting to a total of 16 cores and 32 threads. Each core has two L1 caches of 32 KB (for instructions and data, respectively) and a private 256 KB L2 cache. Cores in each processor share 20 MB of L3 cache. Each processor has 4 memory channels and a peak memory bandwidth of 51.2 GB/s. We used GCC 4.9.2 to compile our OpenMP implementation of the algorithms. We also used `GOMP_CPU_AFFINITY` to request thread pinning in a `scatter` fashion, and `numactl` for NUMA-aware memory allocation. The GPU is a Tesla K40 system, as described in Section 3, consisting of a GK110B GPU with 15 SMXes (2880 streaming processors) at 745 MHz (turbo up to 875 Mhz) and 12 GB of GDDR5 at 6 GHz. We compiled the code using CUDA version 7.0.

Dataset: We experimented with a large dataset of 269 instances (matrices) downloaded from the University of Florida Sparse Matrix Collection [2]. We downloaded matrices that are symmetric and converted the negative nonzero values to positive. For matrices without weights, weights were added uniformly at random between zero and one, and zero-weight edges were discarded. Given a

matrix A of size $m \times n$, we represent each diagonal entry as a vertex. Each off-diagonal entry is represented as an edge between the vertices representing the row and column of that nonzero entry. The nonzero value is set as the weight of that edge. The diagonal entries are ignored. Thus, the graph representing A has $|m|$ vertices, and the number of edges match the number of nonzeros with diagonal entries ignored. In this paper, we present results only for inputs above a million but less than a billion edges. The number of vertices vary based on the sparsity structure of the matrices. We summarize the size distribution in Figure 1. We also experimented with over 300 problems ranging from hundred thousand to a million edges with similar run time behavior. This large set of inputs represents a wide variety of applications and sparsity patterns. Accordingly, we see a wide variation in run time for different algorithms and their variants. For each input, we run each algorithm at least ten times and capture the minimum time among these runs.

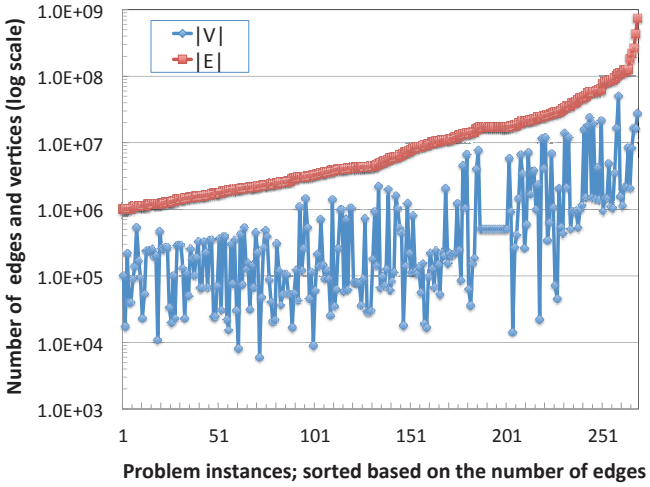

Figure 1: Summary of the sizes of input problems arranged in a non-increasing order of the number of edges. The dataset consists of 269 problems ranging from a million to a billion edges.

5.2 Scaling Comparisons

The two main variants are the Suitor and the Locally-Dominant (LD) algorithms. We implement each algorithm on CPUs using OpenMP (OMP) and on GPUs using CUDA. Thus, we have four main variants to compare: GPU-Suitor, OMP-Suitor, GPU-LD, and OMP-LD. Furthermore, for GPU-Suitor we experiment

with different numbers of threads-per-block and vertices-per-warp as discussed in Section 4. For performance comparisons, we only consider the GPU-Suitor runs with 128 threads-per-block and 8 vertices-per-warp. The impact from variations in these parameters is presented in Section 5.3. Among the four variants presented in Section 4, we present results only for the variant NoSync, the variant with no synchronization and no load balancing. The relative performance of different variants is presented in Section 5.3.

In order to highlight the superior performance of GPU-Suitor, we first present the compute time of GPU and OMP versions of the Suitor and LD algorithms in Figure 2. The run times in milliseconds are presented in log scale on the Y-axis. The times are ordered based on the times of GPU-Suitor. It can be observed that GPU-Suitor outperforms the run time of other variants for most of the problem instances. The OMP run times are for two threads. We present the speedups relative to Suitor and LD algorithms next. The speedup of GPU-Suitor relative

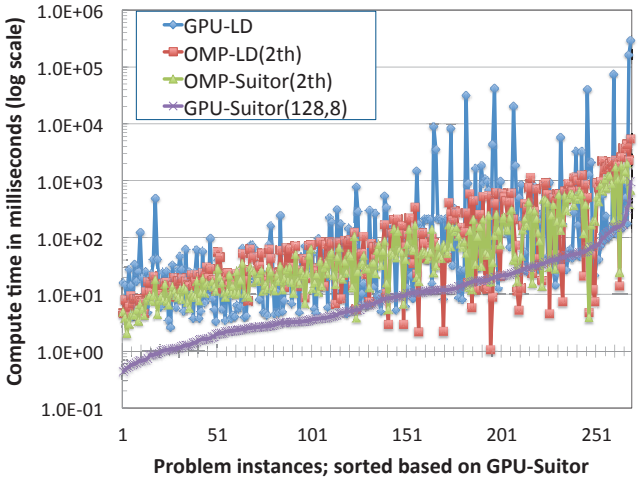

Figure 2: Run time in milliseconds for the two algorithms on two platforms in log scale. The problem instances are shown in non-increasing order of the run times of GPU-Suitor. The OMP times are for two threads.

to GPU-LD and OMP-LD (2 and 16 threads) is presented in Figure 3 on the left, and to OMP-Suitor (2 and 16 threads) on the right. Each speedup curve is ordered individually in non-increasing order of speedup. While we observe positive speedups for GPU-Suitor on a large fraction of the problems against all other algorithms, the largest gains are against GPU-LD. Relative to GPU-LD, the speedups are in the range of 1 to 10× for 115 problems; 10 to 100× for about 100 problems; and above 100× for 15 problems. We run each algorithm for each

input multiple times and pick the minimum time observed among these runs. The experiments were also performed on multiple platforms and we observed similar results. With respect to OMP-Suitor, the best known multithreaded algorithm

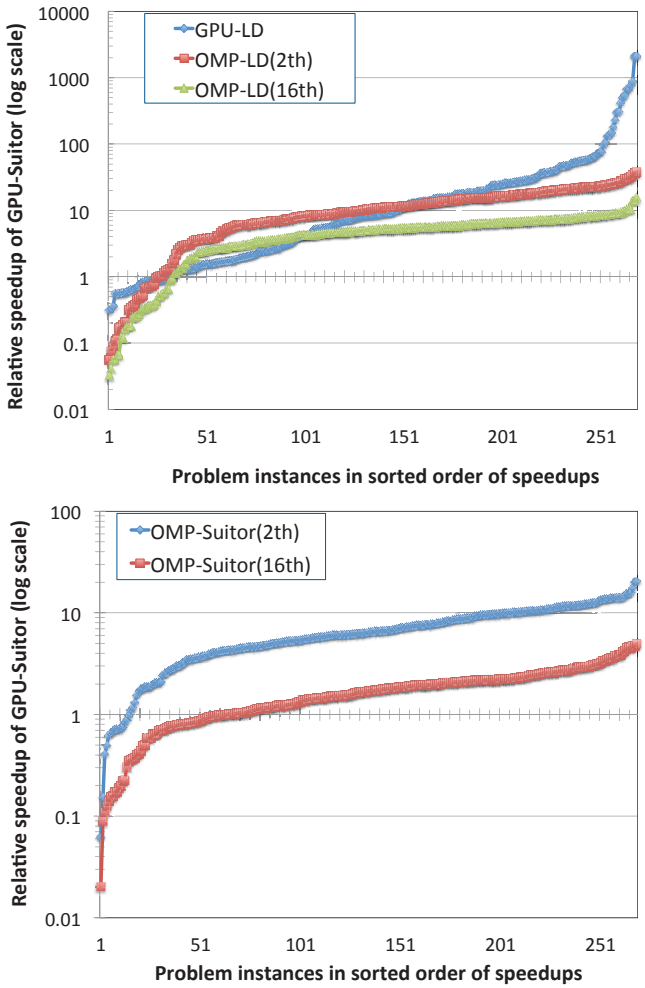

Figure 3: Speedup of GPU-Suitor relative to GPU-LD and OMP-LD on the left, and OMP-Suitor on the right. The speedups are ordered individually for each curve in non-increasing order.

for shared-memory platforms, we observe speedups of up to 20× with two threads

and up to 5× with 16 threads. The speedups with respect to OMP-LD are much higher – up to 40× with two threads and up to 16× with 16 threads.

5.3 Relative Performance

We now present the relative performance of the four variants of GPU-Suitor for different combinations of threads-per-block and vertices-per-warp in this section. The relative performance of variants is presented in Figure 4 in the form of a performance profile. Along the Y-axis we present the fraction of input problems, and along the X-axis we present the relative performance ($\log_2$) to the best variant. For example, we observe that NoSync is the best performing algorithm for about 90% of the problems. However, for about 10% of the problems, NoSync can be up to 4× worse relative to the best variant. We observe that while NoSync stands out as the best variant, the other three variants are similar in performance. The cost of synchronization outweighs the benefits of load balancing.

During the execution of the kernel, most of the blocks finish their work in a few iterations while a few blocks need a significant number of iterations, which in turn determines the overall kernel execution time. If this behavior can be improved without necessitating a large synchronization overhead, GPU-Suitor can perform significantly better.

The second source of difference comes from the variation of threads-per-block and vertices-per-warp. We again present these results in the form of a performance profile captured in Figure 5. We can observe that vertices-per-warp has a large impact on performance, and the threads-per-block has a relatively minor impact. The best performance is obtained with 8 threads-per-warp, and the worst performance is obtained with 256 threads-per-warp, where performance degradation is as high as 30× relative to the best combination. The performance also got worse when less than 8 vertices-per-warp were used.

6 Related Work

Graph algorithm in general, and matching algorithms in particular, are studied extensively. In this section, we present related work that is most relevant to our work. As discussed in Section 2, our work builds on the on multithreaded approximation matching (Locally-Dominant Algorithm) by Halappanavar *et al.* [7], and the Manne and Halappanavar (Suitor Algorithm) [13]. The GPU implementation of Halappanavar *et al.* maintained the general algorithmic structure similar to the implementations on multicore (Intel Xeon) and massively multithreaded (Cray XMT) architectures. In their implementation, the CPU initiates the kernel call considering the number of eligible vertices enqueued in a queue data structure. The actual computation of finding a locally-dominant edge and subsequent

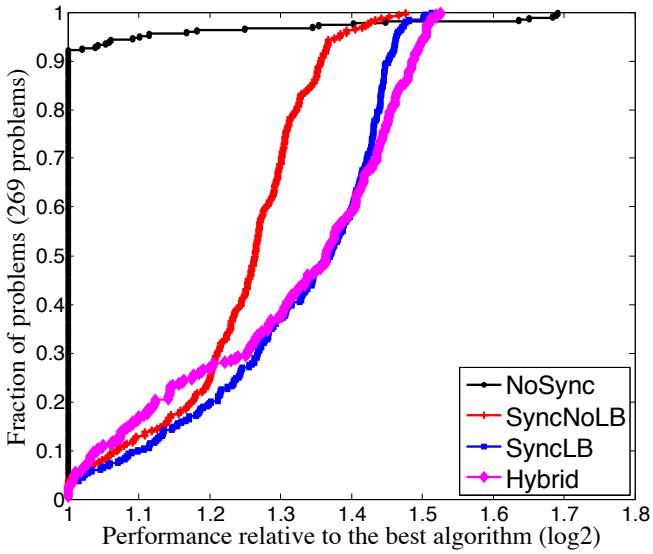

Figure 4: Performance profile depicting the relative performance obtained by different variants of GPU-Suitor. Fraction of input problems are plotted on the Y-axis, and the performance ($\log_2$ scale) relative to the best algorithm are plotted along the X-axis.

matching is done on the GPUs. Matched vertices are concurrently enqueued in a queue for processing in the next iteration. As reported in [8], the increased performance of atomic operations in Fermi-based GPUs provided significant speed ups with respect to a previous generation of hardware. In contrast to the work of Halappanavar *et al.*, we adapt the algorithm of Manne and Halappanavar in this work, which is superior in performance [13]. Further, we consider different combinations of vertices-per-warp and threads-per-block for four variants of the algorithm. The utilization of shared memory is also new in our implementation.

Vasconcelos and Rosenhahn presented GPU adaptation of Bersekas's auction-based algorithm in [20]. However, their implementation is adapted for maximum (unweighted) matching and is limited to bipartite graphs. Fagginger Auer and Bisseling adapt the work of Vasconcelos and Rosenhahn to general graphs by implicitly finding a bipartite graph based on randomly coloring the eligible vertices blue or red [6]. While the blue vertices try to match with one of the neighboring red vertices by bidding, the red vertices select only one bid from the received bids. There are several limitations to this approach, which is not suitable for weighted matching. In our experiments, we found that the quality (in terms of the weight)

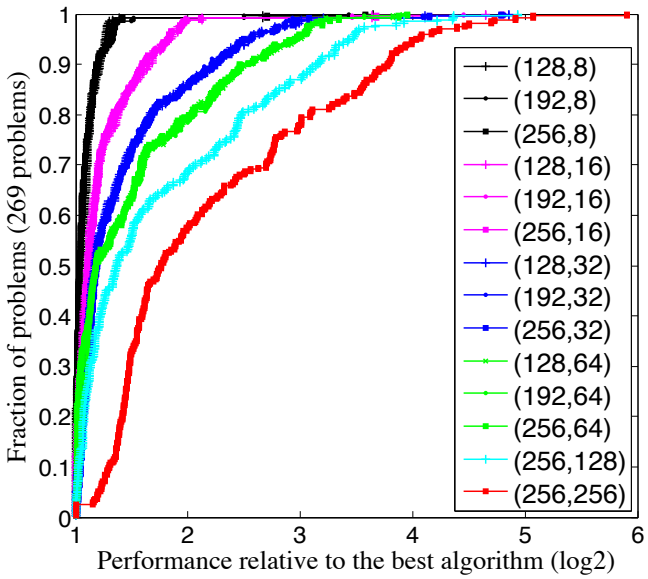

Figure 5: Performance profile depicting the relative performance obtained by different combination of threads-per-block $(128, 192, 256)$ and vertices-per-warp $(8, 16, 32, 64, 128, 256)$. Fraction of input problems are plotted on the Y-axis, and the performance ($\log_2$ scale) relative to the best algorithm are plotted along the X-axis.

of the matching computed with this approach was significantly lower relative to our algorithms. We also note that the algorithm of Auer and Bisseling repeatedly considers all the vertices and is therefore not (work) efficient, but scales better. Xu *et al.* use the algorithm of Auer and Bisseling, along with several other graph algorithms [21]. We address some of the performance issues raised by them in our work. In a similar vein, Devici *et al.* also present adaptation of maximum matching on GPUs [3].

A recent unpublished work of Cohen *et al.* is also relevant to our work [1]. Using a *hand-shaking approach*, Cohen *et al.* identify locally-dominant edges similar to the approach used in Halappanavar *et al.* They further adapt this algorithm by enabling k-way handshake that builds a subgraph by restricting the maximum degree of any vertex to k (k top neighbors of a vertex). However, the performance gain from this extension is not observed in all the inputs. Further, their implementation is specific to bipartite graphs.

Our work benefited from the work of Hong *et al.* that introduced the notion of utilizing the threads of a warp to process the neighborhood of a vertex [9]. As

discussed in several parts of this paper, we present the benefits of this approach over the thread-per-vertex approach of Halappanavar *et al.* Considerable amount of literature exists on implementations of other graph kernels such as breadth-first search, single-source shortest-path, graph coloring and betweenness centrality on modern GPU platforms. We again refer to the work of Xu *et al.* on this topic. An important area of relevant work is on multi-GPUs. While we restricted our focus on a single GPU in this work, we plan to explore multi-GPU implementations in the near future. We refer the work of Mastrostefano and Bernaschi on distributed multi-GPU implementations of the breadth-first algorithm [14].

We conclude this section by noting that to the best of our efforts, this is the first extensive work on implementing the current best approximate matching algorithm of the current best GPU platform using an exhaustive set of variations and input problems.

7 Conclusions

Using weighted matching as a case study, we presented different strategies to exploit GPU architectures such as coaleased memory access, minimizing thread divergence and load balancing. Supported by experimental results we demonstrated not only excellent scaling on the Nvidia Kepler K-40 platform, but also competitive performance relative to traditional multi-core architectures. We demonstrated speedups relative to previous best GPU algorithm by 10 to 100× for over 100 instances, and from 100 to 1000× for 15 instances. We also demonstrated up to 20× speedup relative to 2 threads, and up to 5× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm. We showed the impact of algorithmic variations such as synchronization and load balancing on performance. We also showed the impact of different combinations of threads-per-block and vertices-per-warp on performance.

We conclude this paper by observing that as power limitations impose severe restrictions on architecture design, driving future systems toward larger numbers of weaker cores, this work on a prototypical irregular application (graph algorithm) demonstrates promise of better performance on future low-power architectures. We believe that the algorithmic ideas presented in this paper that exploit architectural features will benefit other researchers implementing their applications on manycore architectures, and that the lessons learned will be applicable to future generations of architectures and other graph algorithms.

Acknowledgment

A part of this work was supported by DoD under project 63810 and the Center for Adaptive Super Computing Software Multithreaded Architectures (CASS-

MT) at the U.S. Department of Energy Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. We thank Oreste Villa for lively discussions and access to previous GPU implementation of matching algorithms.

References

[1] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the GPU. Presentation at NVIDIA GTC conference, 2012.

[2] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[3] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. *CoRR*, abs/1303.1379, 2013.

[4] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[5] Florent Duguet. Kepler vs Xeon Phi : Nos mesures - et leur code source complet. `http://www.hpcmagazine.fr/en-couverture/kepler-vs-xeon-phi-nos-mesures`, June 2013.

[6] BasO. Fagginger Auer and RobH. Bisseling. A gpu algorithm for greedy graph matching. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2012.

[7] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perf. Comput. App.*, 26(4):413–430, 2012.

[8] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perform. Comput. Appl.*, 26(4):413–430, November 2012.

[9] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[10] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, New York, NY, USA, 1995. ACM.

[11] Arif M. Khan, David F. Gleich, Alex Pothen, and Mahantesh Halappanavar. A multithreaded algorithm for network alignment via approximate matching. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 64:1–64:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] L. Lovasz. *Matching Theory (North-Holland mathematics studies)*. Elsevier Science Ltd, 1986.

[13] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 519–528, Washington, DC, USA, 2014. IEEE Computer Society.

[14] Enrico Mastrostefano and Massimo Bernaschi. Efficient breadth first search on multi-gpu systems. *J. Parallel Distrib. Comput.*, 73(9):1292–1305, September 2013.

[15] Burkhard Monien, Robert Preis, and Ralph Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Comput.*, 26(12):1609–1634, 2000.

[16] Ali Pinar, Edmond Chow, and Alex Pothen. Combinatorial algorithms for computing column space bases that have sparse inverses. *Electronic Transactions on Numerical Analysis*, 22:122–145, 2006.

[17] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990.

[18] E. Jason Riedy, Henning Meyerhenke, David Ediger, and David A. Bader. Parallel community detection for massive graphs. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 286–296, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Alexander Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[20] Cristina Nader Vasconcelos and Bodo Rosenhahn. Bipartite graph matching computation on gpu. In *Proceedings of the 7th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition*, EMMCVPR '09, pages 42–55, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Qiumin Xu, Hyeran Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 140–149, Oct 2014.