Performance-Portable Graph Coarsening for Efficient Multilevel Graph Analysis

Michael S. Gilbert*, Seher Acer†, Erik G. Boman†, Kamesh Madduri*, Sivasankaran Rajamanickam†
**Pennsylvania State University*, University Park, USA, {msg5334, madduri}@psu.edu
†*Sandia National Laboratories*, Albuquerque, USA, {sacer, egboman, srajama}@sandia.gov

***Abstract*—The multilevel heuristic is an effective strategy for speeding up graph analytics, and graph coarsening is an integral step of multilevel methods. We perform a comprehensive study of multilevel coarsening in this work. We primarily focus on the graphics processing unit (GPU) parallelization of the Heavy Edge Coarsening (HEC) method executed in an iterative setting. We present optimizations for the two phases of coarsening, a fine-to-coarse vertex mapping phase, and a coarse graph construction phase. We also express several other coarsening algorithms using the Kokkos framework and discuss their parallelization. We demonstrate the efficacy of parallelized HEC on an NVIDIA Turing GPU and a 32-core AMD Ryzen processor using multilevel spectral graph partitioning as the primary case study.**

***Index Terms*—coarsening, multilevel, graph construction, GPU, partitioning**

I. Introduction

The multilevel heuristic [1] is pervasive in large-scale graph analysis. Its applications include graph partitioning [2]–[4], clustering [5]–[7], drawing [8], [9], and representation learning [10], [11]. The family of algebraic multigrid methods [12], [13] in linear algebra is closely related to multilevel methods for graph analysis. Sparse matrices are essentially weighted graphs. Informally, in a multilevel method, instead of solving a problem on a large graph, we build a hierarchy of graphs that are progressively smaller than the original graph and yet preserve the structure of the original graph. We then solve the problem on the smallest graph and *project* or *interpolate* the solution to the original graph using the hierarchy. In multigrid for linear systems, there is also a phase where the residual is *restricted* from a fine graph/matrix to a coarser graph/matrix. Graph coarsening refers to constructing the hierarchy of graphs, and coarsening is the first step in a multilevel method.

While coarsening can be problem-specific, the desirable features of a coarsening strategy remain the same: (i) the solution projected to the original graph must be close to a solution obtained on the original graph, (ii) projecting the solution from the coarsest graphs to the original graph must be fast, and (iii) the cost of the multilevel method must be significantly lower than solving the problem on the original graph. The multilevel heuristic is therefore well suited for NP-Complete problems, though it can also be useful to speed up polynomial time algorithms. Ideally, the cost of coarsening must be linear in the graph size, the number of coarsening levels logarithmic, and the coarsening algorithm must not stall. Coarsening algorithms based on maximal matchings in graphs meet several of the above desirable characteristics and are thus the preferred choice in many multilevel methods.

Coarsening has two parts: *mapping*, how to map the fine vertices to coarse vertices, and the coarse graph construction. The mapping is often done by computing a matching (vertex pairs), but we will also explore methods where several fine vertices are mapped to a single coarse vertex (aggregation, clustering). The mapping can significantly affect performance of multilevel methods as it determines the coarse graphs in the hierarchy. The choice of coarse graph construction method does not change the coarse graphs, but it affects overall performance (run time).

We compare different coarsening methods, including the widely used heavy edge matching (HEM) and the more general heavy edge coarsening HEC [14], which allows a higher coarsening ratio.

For fairly regular graphs such as meshes, many coarsening methods work well. However, for graphs with highly skewed distributions, coarsening can be quite challenging. One difficulty is that the coarsening may be either too aggressive (too many fine vertices merged to a single coarse vertex), or too cautious, causing the coarsening process to stall.

We use graph bisection as our example application, but our techniques are very general and apply to coarsening for a wide variety of problems.

The following are the key contributions of this work:

- We present two related shared-memory parallelizations of the Heavy Edge Coarsening (HEC) algorithm.
- We develop Kokkos-based parallel implementations of HEC and four additional coarsening algorithms, shown to be effective in practice for graph partitioning, vertex embeddings, and algebraic multigrid methods.
- We perform a comprehensive evaluation of coarse graph construction strategies. A new graph construction optimization for skewed-degree graphs may be applicable in other contexts.
- We evaluate coarsening time-quality tradeoffs using multilevel graph bisection as the primary case study.
- Results on a collection of 20 graphs and two architectures (a 32-core AMD CPU and an NVIDIA Turing GPU) indicate that the parallel HEC algorithm consistently outperforms other coarsening strategies on both the CPU and GPU, parallel HEC on the GPU is 2.4× faster than the 32-core CPU, and graph bisection using HEC and Fiduccia-

Algorithm 1 Multilevel Graph Coarsening.

```
Input: Undirected and connected $G(V,E,W) = G_0$. $n = |V|$. Coarsening cutoff $C$.
Output: A set of coarse graphs $\{G_1, \dots, G_l\}$.
1: $i \leftarrow 0$, $n_0 \leftarrow n$
2: while $n_i > C$ do
3:     $i \leftarrow i + 1$
4:     $M_i, n_i \leftarrow$ FINDCOARSEMAPPING($G_{i-1}, n_{i-1}$)
5:     $G_i \leftarrow$ CONSTRUCTCOARSEGRAPH($G_{i-1}, M_i, n_i$)
```

Mattheyses (FM) refinement results in 24% better edge cuts on average compared to the mt-Metis partitioner.

II. Background and Prior Work

We denote a graph as $G(V, E, W)$, where $V = \{1, \dots, n\}$ is the set of vertices, and E the set of edges. $m = |E|$. We assume the graph is undirected, has no self-loops or parallel edges, and has positive edge weights. We assume a compressed sparse row (CSR) storage format. Algorithm 1 gives a high-level overview of graph coarsening in a multilevel method. We can decouple each coarsening iteration into two steps, a fine-to-coarse vertex mapping step (denoted FINDCOARSEMAPPING) and a graph construction step that uses the mapping array M from FINDCOARSEMAPPING and the fine graph G_{i-1} as input. The sequential time of all the algorithms in this section is $O(m + n)$.

The *heavy edge matching* (HEM) algorithm for weighted graphs (Algorithm 2) is a widely used coarsening approach, proposed in the context of graph partitioning [4]. The coarsening ratio, which is the ratio of the vertex count in the fine graph to the count in the coarse graph, is at most two for any matching-based coarsening strategy. Heavy edge coarsening (HEC) is a related coarsening method designed in the context of a multilevel method for computing the Fiedler vector [14]. The pseudocode is given in Algorithm 3. Like HEM, vertices are visited in random order. For each vertex, its heaviest neighbor is identified. If this neighbor has already been mapped, the vertex being considered joins this aggregate. Since HEC is not based on matchings, the coarsening ratio can be arbitrarily high.

mt-Metis [15] is a multilevel graph partitioner for multicore systems, based on the Metis graph partitioner [4]. In [16], LaSalle et al. describe several performance optimizations for graphs with skewed vertex degree distributions. They also present a new coarsening method to mitigate *stalling* seen in HEM-based coarsening. This method uses *two-hop matches*, which are contractions of vertices that are not directly connected, but connected via an intermediary vertex. Among two-hop matches, they specify three sub-classes: leaves, twins, and relatives. After HEM is executed, if the ratio of unmatched vertices to total vertices is greater than some threshold, then leaf, twin, and relative matches are performed. We design GPU algorithms for leaf, twin, and relative matching in this work, while the original work is only focused on CPUs. Stall-free execution is also emphasized in Mongoose [17].

Algorithm 2 Coarsening using Heavy Edge Matching (HEM).

```
Input: $G(V,E,W)$, $n = |V|$.
Output: A mapping array $M[1..n]$, where $M[u]$ is the coarse graph vertex identifier of vertex $u \in V$. Number of coarse vertices $n_c$.
1: $P[1..n] \leftarrow$ GENPERM($n$)
2: $M[1..n] \leftarrow 0$, $n_c \leftarrow 1$
3: for $i \leftarrow 1$ to $n$ do
4:     $u \leftarrow P[i]$
5:     if $M[u] = 0$ then
6:         $w \leftarrow 0$
7:         for each $v$ adjacent to $u$ do
8:             if $M[v] = 0$ and $W(u,v) > w$ then
9:                 $w \leftarrow W(u,v)$
10:                $x \leftarrow v$
11:        if $w > 0$ then
12:            $M[x] \leftarrow n_c$
13:        $M[u] \leftarrow n_c$
14:        $n_c \leftarrow n_c + 1$
```

Algorithm 3 Heavy Edge Coarsening (HEC) algorithm.

```
Input: Connected $G(V,E,W)$. $n = |V|$.
Output: $M[1..n]$, $n_c$.
1: $P[1..n] \leftarrow$ GENPERM($n$)
2: $M[1..n] \leftarrow 0$, $n_c \leftarrow 1$
3: for $i \leftarrow 1$ to $n$ do
4:     $u \leftarrow P[i]$
5:     if $M[u] = 0$ then
6:         $w \leftarrow 0$
7:         for each $v$ adjacent to $u$ do
8:             if $W(u,v) > w$ then
9:                 $w \leftarrow W(u,v)$
10:                $x \leftarrow v$
11:        if $M[x] = 0$ then ▷ $x$ always exists because $G$ is assumed to be connected.
12:            $M[x] \leftarrow n_c$
13:            $n_c \leftarrow n_c + 1$
14:        $M[u] \leftarrow M[x]$
```

We also evaluate two coarsening techniques based on maximal independent sets. Bell et al. [18] design a distance-2 maximal independent set (MIS)-based coarsening technique termed MIS2 for GPUs. The coarse aggregates chosen are a subset of the fine graph vertices such that no two aggregates are within a distance of two of each other. The remaining vertices can be mapped to these coarse aggregates. A related coarsening strategy was recently proposed in the context of a multilevel representation learning technique called GOSH [11]. GOSH is based on MIS (see Algorithm 7 in an extended version of this submission [19]), but with a small change to prevent two high-degree vertices from mapping to each other. In Fig. 1, we show the graphs produced after one level of coarsening using different methods.

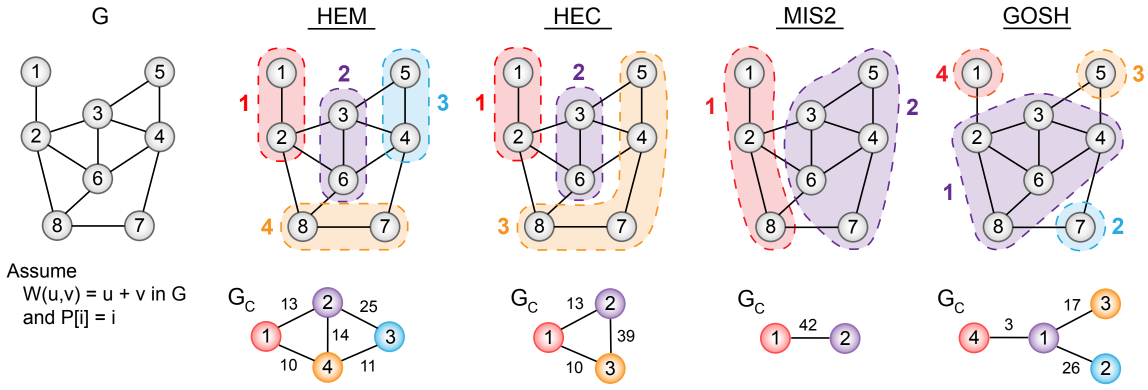

Fig. 1: Coarse graphs produced after one level of coarsening using different methods.

The coarsening methods discussed so far can be termed strict aggregation schemes [20], as they only permit many-to-one mappings of coarse to fine vertices. Weighted aggregation schemes, such as the coarsening strategy in ACE [8] (Algorithm 8 of [19]), allow many-to-many mappings. In preliminary experiments, we found that ACE coarsening quickly makes the coarse graphs dense, and changes to preserve sparsity are left for future work.

Given the mapping vector and a fine graph, there are two main approaches to construct the coarse graph: one based on sorting and the other on hashing. In a global sort-based approach, edge triples $\langle M[u], M[v], W(u,v)\rangle$ are sorted and deduplicated to update weights, whereas in a hashing-based approach, a hash table is built with $\langle M[u], M[v]\rangle$ as the key. We explore vertex-based sorting and hashing methods, where the edge triples are first binned according to the start vertex $M[u]$. An alternate viewpoint of construction arises from linear algebra. Let $\mathcal{P}$ denote an $n_c \times n$ binary matrix such that $\mathcal{P}(M[u], u) = 1$ for all $u \in V$, and 0 otherwise. Then, it turns out that the coarse graph adjacency matrix $A_c = \mathcal{P}A\mathcal{P}^T$, which requires two products of sparse matrices. We thus explore using sparse matrix-matrix multiplication (SpGEMM) implementations for construction.

We use the Kokkos [21] library extensively in this work. Kokkos offers performance-portability, with performant implementations of key routines for both GPUs and multicore systems. The support for parallel primitives such as reductions, mappings, scans, as well as loop parallelization and hierarchical parallelism simplifies programming. Kokkos Kernels [22] is a library of sparse/dense linear algebra and graph kernels implemented using Kokkos. We use the SpGEMM kernel [23] and functionality such as team-level sorting from Kokkos Kernels.

III. Parallelization

A. Coarsening Algorithms

1) HEC parallelization: The coarsening algorithms discussed in the previous section share many similarities, and so the parallelization strategies for each of them are also similar. First, we note that all of them are inherently sequential due

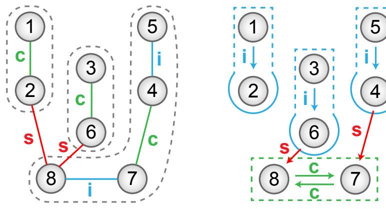

Fig. 2: Illustrating the HEC algorithm. Left: a classification of edges into create, inherit, and skip edges. Right: a directed graph corresponding to the heavy neighbor array H.

to the assumed ordering. This is already known with prior work on HEM-like parallelization [24]. A parallel algorithm would be in the spirit of the sequential algorithm, but will not generally result in the same output as the sequential algorithm.

We will primarily focus on HEC because the algorithm often performs well and is distinct from the rest. Unlike HEM, where a vertex looks for the heaviest *unmatched* neighbor, the heaviest neighbor of a vertex can be predetermined in HEC before entering the matching phase. It is useful to think about a simplified setting where we consider only (up to) n edges, $\langle u, v = H[u]\rangle$. Since the heavy neighbors can be mutual, the size of this edge set is between $n/2$ and n.

One viewpoint of the sequential algorithm is the following. We visit the heavy edge set in some random order, specified by P, and classify each edge in the set as either a *create* edge (where a new coarse vertex is created because u and v are still unmapped), an *inherit* edge (where v is already set and so u inherits the coarse identifier of v), or a *skip* edge (where u is already is set, and so we just ignore this edge). These edges are interspersed, making it difficult for any parallelization to obtain the ordering achieved by the sequential algorithm. Further, a create edge can lead to multiple inherit and skip edges, and an inherit edge can prevent possible new create edges. Thus, it does not appear that we can go out of order. In Fig. 2 (left), we show the labeling of edges upon execution of the sequential HEC algorithm on the graph in Fig. 1.

Our first parallelization of HEC, shown in Algorithm 4, tries to faithfully replicate the sequential algorithm, but with the ordering relaxed. While relaxing the ordering seems like a big assumption, it is necessary in case of GPUs, where there can be tens of thousands of threads in flight at any point. In case of multicore platforms, though, a dynamic scheduling of threads with a small chunk size will be close in spirit to HEC. We avoid locks in Algorithm 4 at the expense of multiple passes. Threads concurrently inspect edges in the heavy edge set and mark both the endpoints (using C for temporary storage) using atomic compare-and-swap instructions. For a create edge (both u and v are still unmapped), only one thread will be able to satisfy both the conditionals and reach line 15 of the algorithm. Skip edges are handled in line 13 with the atomic compare-and-swap on $C[u]$, and inherit edges are processed in line 19. An inherit edge may fail, in which case the thread releases ownership of u. One detail not shown in the algorithm is the case of *mutual* heavy neighbors and duplicate heavy edges (i.e., u's heavy neighbor is v and v's heavy neighbor is u). There is an additional check using vertex identifiers prior to line 13 to prevent deadlocks. The drawbacks of this algorithm are the possible multiple passes (line 29), contention for atomics (particularly line 14, because many vertices can share the same heavy neighbor), and irregular memory references (indirection because of P and H). In practice, the vast majority of vertices are processed in the first two passes. Contention for C is low because of the distribution of edge weights. The irregular memory accesses are also unavoidable. The fast atomics on GPUs also help the implementation of this algorithm. Crucially, one advantage of HEC over HEM is that the edge processing loop (lines 3-8) is relatively simple and does not involve much indirection or fine-grained synchronization. Overall, this algorithm allows for interspersed create, inherit, and skip edges, mirroring the sequential algorithm.

We also devise an alternate parallelization of HEC where we further decouple creating coarse vertices from inherit and the skip edges. To explain the algorithm (Algorithm 5), consider the graph in Fig. 1, and the directed graph induced by the corresponding heavy edge set, shown in Fig. 2 (right). All vertices in this graph have an out-degree of 1 and there are n edges, making this graph a pseudoforest. We mark vertices of non-zero in-degree in this directed graph as coarse vertices. Additionally, we collapse mutual heavy edges in a separate loop (see lines 5-8). Separating the coarse vertex creation step simplifies bookkeeping, and we no longer need the C array. The algorithm also requires very little fine-grained synchronization. We refer to this approach as HEC3. We also considered another approach that is an intermediate step between Algorithm 4 (HEC) and HEC3. This approach, termed HEC2 (see Algorithm 9 of [19]), uses two arrays X and Y to prevent races and consistently assign coarse vertex identifiers. The 2-cycle detection loop of HEC3 is also missing.

2) HEM, Two-hop matching, and GOSH parallelization: We next consider HEM parallelization, which is closely modeled after Algorithm 4. The main distinction between our parallelization of HEM and Algorithm 4 is that the heaviest neighbor is chosen from unmatched vertices. One way to accomplish this is to recompute H for unassigned vertices

Algorithm 4 Lock-free Parallelization of HEC.

```
Input: Undirected and connected $G(V, E, W)$. $n = |V|$.
Output: $M[1..n]$, $n_c$.
 1: $P[1..n] \leftarrow$ ParGenPerm($n$)            ▷ parallel sort-based
 2: $H[1..n] \leftarrow 0$
 3: for $u \leftarrow 1$ to $n$ in parallel do
 4:     $w \leftarrow 0$
 5:     for each $v$ adjacent to $u$ do
 6:         if $W(u,v) > w$ then
 7:             $w \leftarrow W(u,v)$
 8:             $H[u] \leftarrow v$
 9: $C[1..n] \leftarrow 0$, $M[1..n] \leftarrow 0$, $Q \leftarrow P$, $n_c \leftarrow 1$
10: for $i \leftarrow 1$ to $|Q|$ in parallel do
11:     $u \leftarrow Q[i]$, $v \leftarrow H[u]$
12:     if $C[u] = 0$ then
13:         if AtomicCAS($C[u], 0, v$) $= 0$ then
14:             if AtomicCAS($C[v], 0, u$) $= 0$ then
15:                 $m \leftarrow$ AtomicIncr($n_c$)
16:                 $M[u] \leftarrow m$, $M[v] \leftarrow m$
17:             else
18:                 if $M[v] \neq 0$ then
19:                     $M[u] \leftarrow M[v]$
20:                 else
21:                     $C[u] \leftarrow 0$
22: $R[1..|Q|] \leftarrow 0$
23: for $i \leftarrow 1$ to $|Q|$ in parallel do
24:     $u \leftarrow Q[i]$
25:     if $M[u] = 0$ then
26:         Atomically add $u$ to $R$
27: if $|R| > 0$ then
28:     $Q \leftarrow$ NonZeroEntries($R$)
29:     go to line 10
```

after each pass. The pseudocode is given in Algorithm 10 of [19]. In addition to HEM, we will compare to approximation algorithms for weighted maximal matching such as Suitor [25] in future work.

We next design parallel algorithms for two-hop matching in the optimized version [16] of mt-Metis [15]. Two-hop matching is selectively performed, and we use the thresholds used by mt-Metis. Algorithms 11, 12, and 13 in [19] list the pseudocodes for finding leaves, twins, and relatives, respectively. The three approaches are all different, but we can use ideas from HEC and HEM parallelization. Twins are found only if the coarsening threshold is not met after finding leaves, and relatives are found only if the threshold is not met after finding twins.

GOSH coarsening and MIS2 coarsening are related since they are both based on extensions to MIS algorithms. GOSH prevents contractions of two high-degree vertices by design, but if we ignore this detail, then it can be viewed as an MIS algorithm. The MIS is comprised of "u" vertices (see Algorithm 7 in [19]). Another distinction in GOSH is that it uses an ordering of vertices sorted by decreasing degree.

Algorithm 5 An alternate parallelization of HEC's 2nd phase.

```
Input: $G(V,E,W)$. $n = |V|$. $H[1..n]$.
Output: $M[1..n]$, $n_c$.
 1: ($P$ and $H$ are computed as in Alg. 4.)
 2: $M[1..n], O[1..n] \leftarrow 0$
 3: for $i \leftarrow 1$ to $n$ in parallel do
 4:     $O[P[i]] \leftarrow i$                    ▷ $O$ is the inverse of $P$
 5: for $u \leftarrow 1$ to $n$ in parallel do
 6:     $v \leftarrow O[H[u]]$
 7:     if $O[H[v]] = u$ then
 8:         $M[u] \leftarrow \min(u, v)$
 9: for $u \leftarrow 1$ to $n$ in parallel do
10:     $v \leftarrow O[H[u]]$
11:     if $M[v] = 0$ then         ▷ Check to reduce atomic calls
12:         AtomicCAS($M[v], 0, v$)       ▷ AtomicCAS avoids
    unnecessary random writes
13: for $u \leftarrow 1$ to $n$ in parallel do
14:     $v \leftarrow O[H[u]]$
15:     if $M[u] = 0$ then
16:         $M[u] \leftarrow M[v]$
17: for $u \leftarrow 1$ to $n$ in parallel do
18:     $p \leftarrow M[u]$
19:     while $M[p] \neq p$ do
20:         $p \leftarrow M[M[p]]$
21:     $M[u] \leftarrow p$
22: $M, n_c \leftarrow$ FINDUNIQANDRELABEL($M, n$)
```

Algorithm 6 Parallel Coarse Graph Construction.

```
Input: $G(V,E,W)$, $n = |V|$, $m = |E|$, $M$, $n_c$.
Output: $G(V_c, E_c, W_c)$.
 1: $C'[1..n_c] \leftarrow 0$, $C[1..n_c] \leftarrow 0$
 2: for $u \leftarrow 1$ to $n$ in parallel do
 3:     for each $v$ adjacent to $u$ do
 4:         if $M[u] \neq M[v]$ then
 5:             AtomicIncr($C'[M[u]]$)
 6: for $u \leftarrow 1$ to $n$ in parallel do
 7:     for each $v$ adjacent to $u$ do
 8:         if $M[u] \neq M[v]$ then
 9:             if ($C'[M[u]] < C'[M[v]]$) or ($C'[M[u] =$
    $C'[M[v]$ and $u < v$) then
10:                 AtomicIncr($C[M[u]]$)
11: $R \leftarrow$ PARPREFIXSUMS($C$)
12: $m' \leftarrow R[n_c]$, $C[1..n_c] \leftarrow 0$
13: $F[1..m'] \leftarrow 0$, $X[1..m'] \leftarrow 0$
14: for $u \leftarrow 1$ to $n$ in parallel do
15:     for each $v$ adjacent to $u$ do
16:         if $M[u] \neq M[v]$ then
17:             if ($C'[M[u]] < C'[M[u]]$) or ($C'[M[u] =$
    $C'[M[v]$ and $u < v$) then
18:                 $l \leftarrow$ FINDLOC($R, C, u$)
19:                 $F[l] \leftarrow M[v]$, $X[l] \leftarrow W(u, v)$
20: for $u \leftarrow 1$ to $n_c$ in parallel do
21:     $F, X, C[u] \leftarrow$ DEDUPWITHWTS($F, X, R, C, u$)
22: $E_c, W_c, m_c \leftarrow$ GRAPHCONSWITHTRANS($F, X, R, C'$)
```

We implement the MIS(2) algorithm by Bell et al. [18] using Kokkos (pseudocode given in Algorithm 14 of [19]). Our first parallelization of GOSH is based on the MIS(2) parallelization (pseudocode in Algorithm 15 of [19]). One drawback of GOSH is that edge weights are not considered in the mapping process. To rectify this issue, we combine ideas from HEC and HEM parallelizations to develop a new coarsening approach, given in Algorithm 16 of [19]. This alternate approach has less indirection, lower fine-grained synchronization, and skips high-degree vertex adjacencies in several loops.

B. Graph Construction

Algorithm 6 gives the template for vertex-centric graph construction that is common to all the coarsening schemes. The output is a weighted coarse graph, and we assume that both the input and output graphs are required to be in the compressed sparse row (CSR) format. We further assume that the input graph is read-only and the array M uses coarse vertex identifiers from 1 to n_c. The algorithm has six steps. In the first step, we estimate an upper bound on the coarse vertex degree, given by $C'[M[u]]$ (line 5). We omit counting self-loops. Before deduplication, we use an optimization that is specific for undirected graphs. We note that an edge $\langle x, y\rangle$ is stored twice in the CSR format, once in x's adjacency array and again in y's array. However, while deduplicating, we can use just one end. For graphs with skewed distributions, it is preferable to pick the end with lower degree. Since the vertex degree in the coarse graph is not known yet, we use the estimates given by C' to decide which end to store the adjacency. Ties are broken using vertex identifiers. We note that this optimization has been explored for linear algebra-based triangle counting [26], where there is work associated with every edge and it helps to use the low-degree end. Further, observe that sorting vertices by degree is not required. The second step of the algorithm populates the array C to track this count (line 10). In the third and fourth steps, we write the adjacencies and corresponding weights to two intermediate arrays F and X, also organized in CSR format. Per-vertex deduplication now happens in the fifth step using the DEDUPWITHWTS routine. We use two approaches: a bitonic/radix (GPU/CPU respectively) sort-based approach with coarse vertex identifiers as keys and weights as values, followed by striding through the sorted array for in-place deduplication; a hashing-based approach where we use per-vertex hash tables to insert vertex-weight pairs and increment weights. A segmented global sort is also an alternative to separate per-vertex sorts. The sort is better when the duplication factor is close to 1 and for low-degree vertices, whereas hashing is preferable in case of high duplication. A hybrid approach, deciding whether to sort or hash on a per-vertex basis is also possible, and deduplicating coarse adjacencies of each fine vertex is also an additional optimization. We will explore these two optimizations in future work. The final step of graph construction is to enumerate the $\langle v, u\rangle$ edges and store them in the CSR structures for the coarse graph. Since the degree-based deduplication optimization is primarily targeting graphs with skewed degree distributions, we use the ratio of maximum degree to average vertex degree to estimate the skew, and selectively invoke this optimization.

An alternate strategy to coarse graph construction is to compute the $\mathcal{P}A\mathcal{P}^T$ product by calling the SpGEMM implementations in Kokkos Kernels [22], [23] twice. The algorithm uses a symbolic step to compute the actual number of nonzeros in the coarse graph and computes the CSR graph in a numerical step. A local sparse hashmap accumulator is used to avoid any duplicates. In addition to the SpGEMM-based approach, we also use a global sort-based construction method as the baseline, but found the method not to be competitive with approaches discussed here.

C. *Multilevel Graph Partitioning*

With parallel multilevel graph coarsening, we have a key building block of multilevel graph partitioning. The objective of graph partitioning is to partition the set of vertices into k parts such that the number of edges that are *cut* (i.e., edges going across partitions) is minimized and the partitions are balanced (with n/k vertices each). The high-level template for multilevel partitioning is outlined in Algorithm 17 (of [19]), and partitioning methods differ in the algorithms used for initial partitioning and multilevel refinement. We experiment with two refinement methods in this work: one using the eigenvector corresponding to the second-smallest eigenvalue of the graph Laplacian matrix (referred to as *spectral* partitioning), and the other using Fiduccia-Mattheyses (FM) refinement [27]. In case of spectral partitioning, we use the power iteration technique to compute the eigenvector, and the main routine in the power iteration is a sparse matrix-vector multiplication (SpMV). We use the SpMV implementation from Kokkos Kernels. Our FM implementation is currently sequential, running on the CPU. We use the greedy graph growing algorithm for initial partitioning with FM, and the eigenvector associated with the coarse graph in case of the spectral method. In both cases, the interpolation step is straightforward and uses the mapping vectors. We only present results for graph bisection in this work and also do not allow for imbalance in partitions when reporting edge cut. Spectral partitioning is closely related to spectral drawing (where two eigenvectors are used as coordinates for vertices) and spectral clustering (where the balance constraint is relaxed). In future work, we plan to use our new coarse mapping and/or graph construction methods in place of the coarsening routines in well-known multilevel methods for graph clustering, embedding, and other applications.

IV. Empirical Evaluation

We choose a diverse set of graphs to evaluate the coarsening strategies on. The graphs are listed in Table I. We intentionally include just one graph from each application domain, and synthetic graphs constructed using different algorithms. The number of vertices, edges, and average vertex degree span a wide range. Using the ratio of maximum degree to average degree as a measure of degree skew, we split the graphs into two groups (regular and irregular) of ten each. We include three graphs from the Open Graph Benchmark [28] that are used for graph-based learning tasks, and the rest of the graphs are based on SuiteSparse [29] matrices. We preprocess the graphs to make them undirected and extract the largest connected component. Since we evaluate a large collection of algorithms, each with different memory requirements, we are constrained by available main memory on GPUs. We require at least $48m$ bytes for most programs (details in [19]). The graphs considered are initially unweighted but become weighted after one level of coarsening.

TABLE I: A collection of undirected graphs used for performance evaluation. The graphs are based on sparse matrices from the SuiteSparse matrix collection [29], and networks from OGB [28]. We preprocess the graphs to extract the largest connected component and relabel vertex identifiers. The number of edges (m), number of vertices (n), and the ratio of max vertex degree (Δ) to average degree after preprocessing are given. Based on this ratio, we partition graphs into two groups: regular and skewed-degree. Within each group, the graphs are ordered by size ($2m + n$).

Graph	Domain	m	n	$\Delta/(2m/n)$
HV15R	cfd	162 357 569	2 017 169	3.1
rgg24	syn	132 557 200	16 777 215	2.5
nlpkkt160	opt	110 586 256	8 345 600	1.0
europeOsm	road	54 054 660	50 912 018	6.1
CubeCoup	fem	62 520 692	2 164 760	1.2
delaunay24	syn	50 331 601	16 777 216	4.3
Flan1565	fem	57 920 625	1 564 794	1.1
MLGeer	sim	54 687 985	1 504 002	1.0
cage15	bio	47 022 346	5 154 859	2.5
channel050	sim	42 681 372	4 802 000	1.0
ic04	www	149 054 854	7 320 539	6296.9
Orkut	soc	117 185 083	3 072 441	436.7
vasStokes4M	vlsi	97 708 521	4 344 906	25.3
kmerU1a	bio	66 393 629	64 678 340	17.0
kron21	syn	91 040 839	1 543 901	1813.7
products	ecom	61 806 303	2 385 902	337.4
hollywood09	soc	56 306 653	1 069 126	108.9
mycielskian17	syn	50 122 871	98 303	48.2
citation	cit	30 344 439	2 915 301	480.4
ppa	bio	21 231 776	576 039	44.0

We use the NVIDIA GeForce RTX 2080 Ti GPU based on the Turing architecture in our evaluations. This GPU has 68 streaming multiprocessors (SM), each with 64 INT32 cores. Each SM supports concurrent execution of 32 warps (or a total of 1024 threads). The 11 GB GDDR6 memory offers a theoretical peak bandwidth of 616 GB/s. Turing supports the Independent Thread Scheduling feature introduced in the previous Volta architecture. This GPU is on a system with a 32-core AMD Ryzen Threadripper 3970x processor (based on the "Castle Peak" Zen2 architecture). This processor supports 64 hardware threads and has 256 GB quad-channel DDR4-3200 memory for a peak memory bandwidth of 102.4 GB/s. The operating system is Ubuntu Linux 20.04 LTS. Unless otherwise mentioned, all results indicated as CPU are for 32-core execution. We build our programs with GCC 9.3.0. We use CUDA 10.1 and the GPU driver version is 450.51.05. We use Kokkos 3.1.01 and Kokkos Kernels 3.1.01. On the CPU, we see a bandwidth of 77 GB/s with the STREAM Copy benchmark, and on the GPU, the CUDA bandwidth test reports 532 GB/s as the device-to-device bandwidth.

For most experiments, we perform 10 runs and report the median. We also compute the mean and standard deviation of edge cut, execution time, and other metrics. All coarsening methods use a coarse vertex cutoff of 50. However, if the vertex count drops from greater than 50 to less than 10 in an iteration, we discard the coarsest graph. When using the power iteration to compute the eigenvector for partitioning, we use the difference of the 2-norm of the iterates as the stopping criterion. We stop when the difference is lower than 10^{-10}.

A. HEC and Graph Construction Performance

We first evaluate performance of HEC (Algorithm 4) on the GPU and CPU systems, while varying the graph construction strategy. In Table II, we report the total time spent in multilevel coarsening when using sort-based deduplication in the graph construction, and also the fraction of overall time spent in graph construction. Note that the reported times include initial CPU-GPU data transfer times for the graph structure. For irregular graphs, graph construction makes up a higher fraction of total time than regular graphs. The sort-based graph construction method consistently outperforms alternatives, with the speedups more pronounced for irregular graphs. The performance impact of the degree-based deduplication strategy is not shown in this table, but it is quite effective for irregular graphs. For instance, the graph construction time on kron21 is 25.7× higher without this optimization.

Comparing the three variants of HEC, we observe that HEC is 1.13× faster than HEC3 (Algorithm 5) and 1.21× faster than HEC2. These are with geometric means of running time ratios. kron21 is the only instance for which HEC2 and HEC3 are faster than HEC. Additionally, HEC3 runs out of memory on europeOsm. One reason why HEC is faster is that it requires fewer coarsening levels compared to HEC3 (1.26× more levels on average) and HEC2 (1.56×). Further, most of the vertices are processed in just two passes in Algorithm 4 (99.4% for first level of coarsening across all graphs, and 96.7% for the second level of coarsening), and the performance of atomic compare-and-swap does not appear to be a bottleneck. The advantage of HEC2 and HEC3 over HEC is that the coarse vertex count is more predictable.

Next, we look at CPU performance (32-core execution times) for HEC with different graph construction strategies in Table III. europeOsm and kmerU1a are the sparsest graphs in each category, and the percentage time in the mapping step is much higher than the average. Hasing-based construction is consistently the fastest. For regular graphs, the sort-based approach is faster than the SpGEMM-based graph construction. For regular graphs, the time spent in the mapping step is on average higher than for irregular graphs.

In Fig. 3 (left), we report GPU HEC performance normalized to graph size. There do not appear to be any outliers and the performance rates for the graphs fall within a relatively narrow band. It is not possible to say if one group of graphs outperforms the other. In general, performance seems to be better for larger graphs, which is expected behavior on GPUs. In Fig. 3 (middle), we report speedup achieved by the GPU runs over 32-core CPU runs, both using the default (sorting)

TABLE II: Performance of HEC-based coarsening on the GPU. We report total coarsening time using HEC (t_c), percentage of coarsening time spent in graph construction when using the default sort-based coarsening (denoted % GrCo), and the ratio of the graph construction time with alternate methods (hashing-based and SpGEMM-based) to sort-based graph construction.

Graph	t_c (s)	% GrCo	$t_{\text{GrCo-alt}}/t_{\text{GrCo-sort}}$ Hashing	SpGEMM
HV15R	1.00	51	1.14	2.91
rgg24	0.97	41	1.25	2.13
nlpkkt160	0.91	53	1.52	2.09
europeOsm	1.11	34	1.49	2.46
CubeCoup	0.45	51	1.53	1.84
delaunay24	0.54	36	1.45	2.40
Flan1565	0.42	51	1.52	1.87
MLGeer	0.38	49	1.28	1.71
cage15	0.49	58	1.66	3.08
channel050	0.35	45	1.79	2.03
GeoMean ↑		46	1.45	2.21
ic04	0.90	45	1.26	6.43
Orkut	1.76	78	1.36	3.71
vasStokes4M	0.61	44	1.61	4.22
kmerU1a	1.47	35	1.68	3.38
kron21	1.76	85	1.17	3.44
products	0.48	56	1.77	6.39
hollywood09	0.56	67	1.86	4.14
mycielskian17	0.73	80	1.88	1.87
citation	0.30	58	2.22	7.88
ppa	0.19	57	3.04	6.09
GeoMean ↑		58	1.72	4.41

TABLE III: Performance of HEC-based coarsening on the multicore system. We report total coarsening time using HEC (t_c), percentage of coarsening time spent in graph construction when using the default sort-based coarsening (denoted % GrCo), and the ratio of the graph construction time with alternate methods (hashing-based, SpGEMM-based) to sort-based.

Graph	t_c (s)	% GrCo	$t_{\text{GrCo-alt}}/t_{\text{GrCo-sort}}$ Hashing	SpGEMM
HV15R	1.69	69	0.63	0.68
rgg24	2.01	44	0.70	1.52
nlpkkt160	1.64	60	0.68	1.25
europeOsm	3.40	24	0.97	2.93
CubeCoup	0.73	63	0.66	0.95
delaunay24	1.41	36	0.85	2.06
Flan1565	0.66	65	0.66	0.95
MLGeer	0.59	63	0.65	0.97
cage15	1.00	65	0.68	1.16
channel050	0.67	52	0.68	1.45
GeoMean ↑		52	0.71	1.28
ic04	2.44	72	0.50	0.50
Orkut	2.89	84	0.80	0.78
vasStokes4M	1.32	66	0.64	0.89
kmerU1a	4.78	28	0.97	2.80
kron21	3.46	90	0.96	0.42
products	1.04	74	0.69	1.17
hollywood09	1.11	81	0.85	0.67
mycielskian17	0.86	82	0.90	0.51
citation	0.70	70	0.71	1.37
ppa	0.40	75	0.80	0.97
GeoMean ↑		69	0.77	0.86

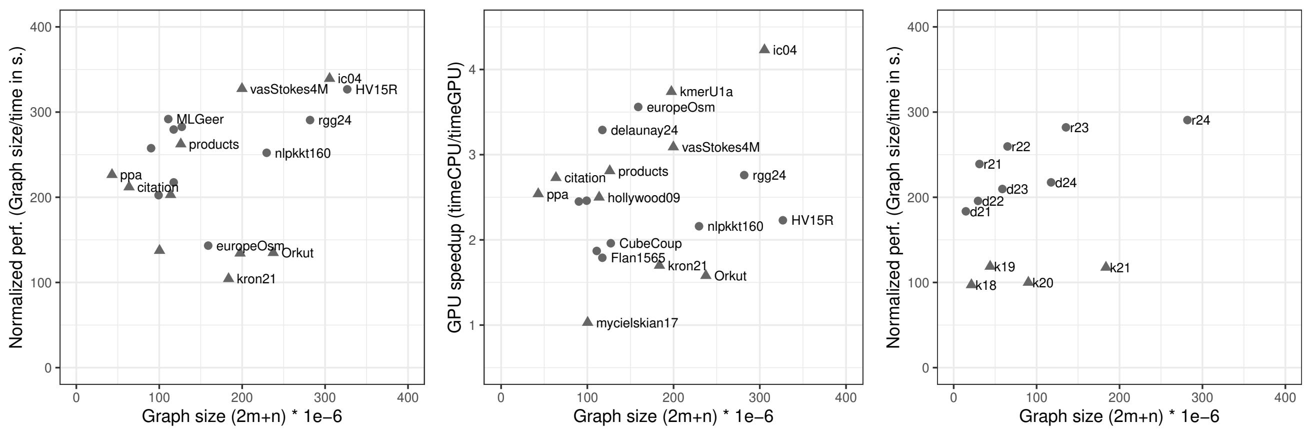

Fig. 3: Performance of HEC-based coarsening. Left: GPU performance rate (running time normalized to graph size), Center: Speedup using GPU, Right: Weak scaling for synthetic graphs (rgg, kron, delaunay).

deduplication. Data transfer times are excluded here. The geometric mean is 2.4×. In Fig. 3 (right), we show weak scaling results for three synthetic graph families: rgg, delaunay, and kron. Performance is generally higher for larger graphs. The regular rgg and delaunay graphs show higher performance than the kron family due to better load balance in adjacency processing steps.

B. Evaluating Coarsening Algorithms

We use sort-based deduplication as the default graph construction method on both CPUs and GPUs in subsequent experiments. Table IV compares GPU performance of various coarse mapping methods. The first part of the table reports ratios of total coarsening time using an alternate matching strategy to the previously-discussed GPU performance numbers for HEC. Values greater than 1 indicate that HEC is faster than the respective method. Looking at geometric means of slowdowns for regular graphs, we see the next fastest methods are MIS2, mtMetis's two-hop matching, HEM, and GOSH. mtMetis can be faster than matching because it can coarsen more aggressively. For irregular graphs, the ordering is GOSH, MIS2, mt-Metis, and HEM. In three instances of irregular graphs, HEM runs out of memory. The second part of the table gives the number of coarsening levels, with lower values indicating more aggressive coarsening. If coarsening happens very rapidly, the output may not be useful (e.g., MIS2 and HEC for mycielskian17). mtMetis coarsening outperforms HEM in all cases, with very high impact on the number of levels in many cases. We further compare HEC and mtMetis coarsening using the average coarsening ratio. mtMetis runs out of memory for two instances, and HEC coarsens the irregular graphs at a considerably higher rate. For irregular graphs, while GOSH requires more levels than MIS2, it is slightly faster overall. This is because MIS2 coarsening performs more work at each level, while GOSH (similar to MIS) is less aggressive in coarsening. We also find that the algorithm based on GOSH and HEC is 1.46× faster than GOSH (across all graphs) and also results in 1.18× lower levels than GOSH. This new strategy could be useful for weighted graphs.

C. Graph Partitioning Results

We now evaluate performance of different coarsening methods for graph bisection. Table V summarizes the results. We first report the time for partitioning and percentage of time spent in coarsening, when using HEC coarsening. From the geometric means, we see that the refinement stage takes longer on irregular graphs than on regular graphs. For ppa and mycielskian17, the time in refinement is more than 10× the time in coarsening. This could either mean that the coarsening is very aggressive or that the underlying matrix is a challenging instance with close eigenvalues. While we can address the former case (aggressive coarsening), the latter is a limitation of the spectral method. We next compare edge cuts by looking at ratios to the cut when using HEC. Ideally, all the ratios must be close to 1. However, we see that this is not the case for several graphs (e.g., vasStokes4M, kmerU1a, ppa). This indicates misconvergence or that the method is not spending adequate time in refinement. Since we use the same stopping criterion for all methods, we suspect misconvergence. For regular graphs, HEC coarsening outperforms all other strategies. The next best approach is mt-Metis, followed by MIS2, and then HEM and GOSH (full results in [19]). It is surprising that MIS2 is quite effective, given that it coarsens aggressively. For irregular graphs, even a few vertex swaps can have considerable impact on edge cut, especially for bisection. We note that the coarsening rate of HEM is quite slow for some of the larger graphs (ic04, Orkut, kron21), and the 11 GB memory on the GPU becomes a bottleneck. While two-hop matching works for the ic04 web crawl, it fails (OOM) for the social network Orkut and kron21, the synthetic graph with high degree skew. There are possibly large near-clique structures in both these graphs preventing faster two-hop matching.

We now devise an alternative multilevel partitioner, using the FM refinement instead of the spectral method. FM is inherently sequential and we are unaware of FM parallelizations

TABLE IV: Comparison of coarsening methods on the GPU. We report the ratio of coarsening time using one of the four alternatives to the coarsening time using HEC (see Table II). We use sort-based graph construction. We also report number of levels (l) for each method and average coarsening ratio ($cr = (n_0/n_l)^{l^{-1}}$) for HEC and mt-Metis coarsening. Runs where the GPU memory was insufficient are indicated as OOM (Out of Memory).

Graph	$t_{\text{c-alt}}/t_{\text{c-HEC}}$				Number of levels l					$cr = (n_0/n_l)^{l^{-1}}$	
	HEM	mtMetis	GOSH	MIS2	HEC	HEM	mtMetis	GOSH	MIS2	HEC	mtMetis
HV15R	2.04	2.05	1.11	1.72	8	17	17	13	3	5.19	1.96
rgg24	1.67	1.65	1.22	0.91	12	30	24	21	6	3.21	1.75
nlpkkt160	1.89	1.90	1.45	1.31	10	21	21	14	4	4.27	1.86
europeOsm	2.04	1.72	26.50	0.96	14	201	29	20	10	3.09	1.66
CubeCoup	1.58	1.59	2.63	1.20	9	18	18	27	4	4.27	1.91
delaunay24	1.88	1.64	1.08	0.94	12	103	25	20	7	3.19	1.71
Flan1565	1.59	1.59	1.30	1.19	9	18	18	13	4	4.08	1.88
MLGeer	1.58	1.57	OOM	1.14	9	18	18	OOM	5	3.84	1.85
cage15	1.97	1.97	1.16	1.03	9	19	19	12	4	4.88	1.96
channel050	1.66	1.67	OOM	0.89	10	20	20	OOM	4	4.01	1.84
GeoMean ↑	1.78	1.73	1.97	1.11						3.95	1.84
ic04	OOM	6.35	2.25	2.02	8	OOM	24	9	6	6.33	1.69
Orkut	OOM	OOM	1.40	1.96	6	OOM	OOM	11	3	7.98	OOM
vasStokes4M	2.04	2.03	3.28	1.38	8	22	21	16	4	5.32	1.80
kmerU1a	1.93	1.67	0.99	1.38	9	201	29	16	8	4.90	1.68
kron21	OOM	OOM	0.88	2.42	3	OOM	OOM	7	3	12.98	OOM
products	2.95	2.33	1.88	3.52	7	201	22	11	4	6.56	1.67
hollywood09	2.14	2.12	1.41	1.47	6	24	19	8	3	8.91	1.77
mycielskian17	1.63	1.80	1.31	0.28	3	16	14	8	1	9.04	1.84
citation	3.24	2.24	1.88	3.52	7	201	23	11	4	6.59	1.67
ppa	4.63	2.58	1.91	2.12	6	201	18	12	3	6.69	1.74
GeoMean ↑	2.50	2.40	1.60	1.70						7.24	1.73

TABLE V: Performance of spectral bisection on GPU with different coarsening methods. We report the total partitioning time using HEC coarsening, the percentage of time spent in coarsening, and the edge cut (median of ten runs). We also report an edge cut ratio ($cut_{\text{alt}}/cut_{\text{HEC}}$) when using an alternate coarsening scheme.

Graph	Time (s)	%Coa	Edge cut	$\text{cut}_{\text{alt}}/\text{cut}_{\text{HEC}}$	
				HEM	mtMetis
HV15R	1.81	55	874 166	0.99	0.99
rgg24	1.40	69	27 062	1.30	0.98
nlpkkt160	3.24	28	555 014	1.16	1.12
europeOsm	1.49	74	685	3.32	2.32
CubeCoup	0.89	51	348 824	1.02	1.04
delaunay24	0.93	58	9650	2.07	0.99
Flan1565	0.63	66	35 514	1.01	1.30
MLGeer	0.52	73	52 993	0.99	0.99
cage15	6.06	8	1 263 238	1.03	1.01
channel050	0.72	49	49 882	1.03	0.98
GeoMean ↑		46		1.27	1.12
ic04	1.09	83	471 776	OOM	1.66
Orkut	15.52	11	16 874 701	OOM	OOM
vasStokes4M	1.86	33	269 564	1.07	0.85
kmerU1a	9.75	15	560 981	0.78	0.78
kron21	6.68	26	44 684 970	OOM	OOM
products	0.65	74	4 427 711	1.65	0.96
hollywood09	2.74	20	7 825 700	1.07	1.01
mycielskian17	7.72	9	22 001 064	1.09	1.08
citation	0.52	59	2 629 260	1.64	0.88
ppa	2.47	8	2 950 901	1.08	0.77
GeoMean ↑		24		1.16	0.97

for massively multithreaded architectures. We use parallel HEC coarsening on the GPU and CPU. In Table VI, we first report edge cut on the GPU. Comparing these cuts to the results in Table V, we see that FM outperforms spectral on 19 of the 20 instances. For regular graphs, the geometric mean of edge cut ratios is 1.29, and for irregular graphs, it is 4.57. This appears to suggest that the spectral method, even with different coarsening strategies, is not a good fit for irregular graphs in terms of cut performance. The irregular graph results are skewed by performance on ic04, Orkut, and kron21. The performance of the FM refinement with parallel HEC coarsening on CPU is slightly better than the GPU version. mt-Metis is better than Metis for all the irregular graphs, but produces a significantly better cut than FM+GPU-HEC for only one instance (vasStokes4M). For the regular graphs, Metis is competitive with mt-Metis. The coarsening optimizations of mt-Metis help for europeOsm. mt-Metis is 2.17× faster than the fully parallel GPU partitioner for regular graphs, and also produces slightly better cuts. For irregular graphs, the advantage of mt-Metis is not that significant over the GPU spectral partitioner (1.12×), but the edge cut improvement is much more pronounced than regular graphs. This does not factor in memory use of mt-Metis, and we note that mt-Metis coarsening failed on the GPU for two irregular instances. We conclude by highlighting the significant edge cut improvement achieved by FM+GPU-HEC over the state-of-the-art mt-Metis shared-memory partitioning technique: 16% for regular graphs and 35% for irregular graphs.

V. Conclusions and Future Work

To summarize, we design and evaluate seven new algorithms: three HEC parallelizations, HEM, mtMetis two-hop matching (new for the GPU), and two GOSH parallelizations. Additionally, we implement the MIS2 coarsening algorithm from [18]. All of them work on both GPUs and multicore CPUs. For graph construction, we have three approaches: vertex-centric deduplication using either sorting or hashing, and SpGEMM-based construction. We also implemented the

TABLE VI: Comparing performance of multilevel graph bisection using FM refinement. Only the coarsening phase is parallelized. We also report edge cuts obtained with mt-Metis v0.7.2 (indicated as mtMts) and Metis v5.1.0 (denoted as Mts). Finally, we report the ratio of the running time for GPU spectral partitioning with HEC coarsening (SpGPU) to mt-Metis 32-core execution time ($t_{\text{Spec+GPU-HEC}}/t_{\text{mt-Metis}}$).

Graph	FM+GPU-HEC	$\text{cut}_{\text{alt}}/\text{cut}_{\text{FM+GPU-HEC}}$				GPU vs.
	Edge cut	FM+CPU	SpGPU	Mts	mtMts	mtMts
HV15R	768 220	0.91	1.14	1.13	1.02	2.98
rgg24	17 358	0.99	1.56	1.52	1.5	1.49
nlpkkt160	514 680	1	1.08	1.07	1.02	3.09
europeOsm	217	0.96	3.16	3.06	1.06	0.47
CubeCoup	333 488	1	1.05	1.02	1.15	3.54
delaunay24	9018	0.98	1.07	1.14	1.16	1.17
Flan1565	35 937	1	0.99	0.99	1.36	3.18
MLGeer	52 024	1	1.02	1.02	1.38	3.25
cage15	721 506	0.92	1.75	1.92	1.03	3.81
channel050	48 836	0.99	1.02	1.02	1.39	2.24
Geomean ↑		0.97	1.29	1.29	1.19	2.17
ic04	46 046	1	10.25	10.55	2.31	0.65
Orkut	614 080	1	27.48	2.28	1.62	1.87
vasStokes4M	224 704	0.97	1.2	1.22	0.86	1.94
kmerU1a	238 539	0.98	2.35	2.33	1.09	1.07
kron21	2 463 150	1	18.14	14.85	9.42	0.43
products	1 115 524	0.97	3.97	1.24	1.13	0.32
hollywood09	1 966 982	1	3.98	3.94	1.19	1.24
mycielskian17	11 970 808	1	1.84	1.84	1.7	8.39
citation	567 094	1	4.64	1.08	1.01	0.29
ppa	1 439 662	1	2.05	1.05	0.99	3.09
Geomean ↑		0.99	4.57	2.52	1.54	1.12

ACE coarsening strategy and a graph construction strategy using heaps for deduplication on the CPU, but do not include results here. Our key contributions are summarized at the end of Section I. We identify several future work directions: designing algorithms that exploit the HEC connection to pseudoforests, evaluating b-matching and the b-Suitor algorithm [30] for coarsening, fully parallel partitioning with FM-based refinement, and use with multi-GPU spectral partititoners [31].

Acknowledgment

This work was supported in part by the U.S. National Science Foundation grant #1955971 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] S.-H. Teng, "Coarsening, sampling, and smoothing: Elements of the multilevel method," in *Algorithms for Parallel Processing*, 1999, pp. 247–276.

[2] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. Supercomputing '95*. ACM, December 1995.

[3] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101–117, 1994.

[4] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, 1998.

[5] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 11, pp. 1944–1957, 2007.

[6] Y. Goldschmidt, M. Galun, E. Sharon, R. Basri, and A. Brandt, "Fast multilevel clustering," Weizmann Institute of Science, Tech. Rep., 2005.

[7] B. F. Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU," *Graph Part. and Graph Clustering*, vol. 588, p. 223, 2012.

[8] Y. Koren, L. Carmel, and D. Harel, "Drawing huge graphs by algebraic multigrid optimization," *Multiscale Modeling & Simulation*, vol. 1, no. 4, pp. 645–673, 2003.

[9] Y. Hu and L. Shi, "Visualizing large graphs," *WIREs Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.

[10] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, "HARP: Hierarchical representation learning for networks," in *Proc. AAAI Conf.*, 2018.

[11] T. A. Akyildiz, A. A. Aljundi, and K. Kaya, "GOSH: Embedding big graphs on small hardware," in *Proc. ICPP*, 2020.

[12] A. Brandt, "Algebraic multigrid theory: The symmetric case," *Applied mathematics and computation*, vol. 19, no. 1-4, pp. 23–56, 1986.

[13] J. Xu and L. Zikatanov, "Algebraic multigrid methods," *Acta Numerica*, vol. 26, p. 591–721, 2017.

[14] J. C. Urschel, J. Xu, X. Hu, and L. T. Zikatanov, "A Cascadic Multigrid Algorithm for computing the Fiedler vector of graph Laplacians," *Journal of Comp. Math.*, vol. 33, no. 2, pp. 209–226, 2015.

[15] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. IPDPS*, 2013, pp. 225–236.

[16] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis, "Improving graph partitioning for modern graphs and architectures," in *Proc. Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2015.

[17] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan, "Algorithm 1003: Mongoose, a graph coarsening and partitioning library," *ACM Trans. Math. Softw.*, vol. 46, no. 1, Mar. 2020.

[18] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[19] M. S. Gilbert, S. Acer, E. G. Boman, K. Madduri, and S. Rajamanickam, "Performance-portable graph coarsening for efficient multilevel graph analysis," 2021, technical report, DOI: 10.26207/mwqw-fb88.

[20] C. Chevalier and I. Safro, "Comparison of coarsening schemes for multilevel graph partitioning," in *Learning and Intelligent Optimization*, Berlin, Heidelberg, 2009, pp. 191–205.

[21] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–16, 2014.

[22] "Kokkos Kernels," 2017, https://github.com/kokkos/kokkos-kernels.

[23] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures," *Parallel Computing*, vol. 78, pp. 33–46, 2018.

[24] B. O. Fagginger Auer and R. H. Bisseling, "A GPU algorithm for greedy graph matching," in *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*, 2012, pp. 108–119.

[25] F. Manne and M. Halappanavar, "New effective multithreaded matching algorithms," in *Proc. IPDPS*, 2014, pp. 519–528.

[26] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *Proc. HPEC*, 2017.

[27] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Design Automation Conference*, 1982, pp. 175–181.

[28] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv:2005.00687*, 2020.

[29] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. on Mathematical Software*, vol. 38, no. 1, 2011.

[30] A. Khan, A. Pothen, M. Mostofa Ali Patwary, N. R. Satish, N. Sundaram, F. Manne, M. Halappanavar, and P. Dubey, "Efficient approximation algorithms for weighted b-matching," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S593–S619, 2016.

[31] S. Acer, E. G. Boman, and S. Rajamanickam, "SPHYNX: Spectral Partitioning for HYbrid aNd aXelerator-enabled systems," in *Proc. Int'l. Parallel and Distributed Proc. Symp. Workshops (IPDPSW)*, 2020.