

A GPU Algorithm for (3, 4)-Nucleus Decomposition via Clique Reduction

Amir Mehrafsa
University of Victoria
Victoria, BC, Canada
mehrafsa@uvic.ca

Sean Chester
University of Victoria
Victoria, BC, Canada
schester@uvic.ca

Alex Thomo
University of Victoria
Victoria, BC, Canada
thomo@uvic.ca

Abstract

Nucleus decomposition uncovers a hierarchy of dense subgraphs by repeatedly peeling cliques according to how many larger cliques contain them. Its (3, 4)-nucleus, which ranks triangles by the number of four-cliques on them, exposes a finer and more cohesive decomposition than the widely used k -core and k -truss, but at higher computational cost, which makes it a natural candidate for GPU acceleration. Yet no GPU algorithm for it has been reported, because the recursive clique enumeration at its core maps poorly onto massively data-parallel hardware. We close this gap. Our central idea is a *clique-reduction* primitive that, applied iteratively, replaces clique enumeration on the input graph with enumeration on a successively reduced graph and linearises the intermediate structures into flat edge, degree, and offset arrays. This layout is amenable to data-parallel GPU execution and lets peeling proceed over contiguous memory, while the reduction itself admits provable linearisability, composability and partitionability properties that we establish. Four-cliques outnumber edges by orders of magnitude on real graphs, and past a few billion they no longer fit in device memory. We therefore also peel *without ever storing them*, recomputing each triangle’s four-cliques on demand from an index that is linear in the number of triangles. We implement the resulting algorithm in CUDA and evaluate it on standard benchmark graphs. Against ARB-NUCLEUS-DECOMP (ARB), the parallel CPU state of the art at its best thread count, our algorithm on one H100 is 3 to 39× faster where the four-cliques fit in device memory. On LiveJournal and Orkut, whose 9.9 and 3.2 billion four-cliques do not fit, recomputation still completes in under a minute, 2.5 and 4.8× faster than ARB, while a recent counting-based method does not finish within an hour.

Keywords

nucleus decomposition, dense subgraphs, GPU algorithms, clique enumeration, peeling

1 Introduction

Dense-subgraph discovery is central to structural graph analysis [8, 9], used in community detection, fraud and anomaly detection, biological network analysis, and web-graph summarisation. The two most widely used density formulations, the k -core and the k -truss, are both special cases of *nucleus decomposition*, which measures density through the support of one clique inside another and so generalises the notion to arbitrary (r, s) -clique pairs [13, 14]. Concretely, the k -core is the (1, 2)-nucleus and the k -truss is the (2, 3)-nucleus. Moving to larger cliques exposes denser and more finely resolved hierarchies that vertex-degree or edge-support measures cannot see, and the (3, 4)-nucleus, which

ranks triangles by the number of four-cliques that contain them, is the canonical higher-order instance and the target of this paper.

The gain in resolution is large and it is practical. Call the density of a subgraph the fraction of its vertex pairs that are joined by an edge. In a protein-interaction network, Esfahani et al. [6] find that the k -core and the k -truss around three proteins of interest span thousands of vertices at density below 0.05, too loose to say anything about the proteins, whereas the (3, 4)-nucleus around the same proteins has 95 vertices at higher density, small and tight enough to interpret biologically. The same effect shows on ordinary benchmarks. On the citation network cit-patent we computed all three decompositions and counted, for each, the subgraphs with at least five vertices and density at least 0.8.

decomposition	subgraphs (≥ 5 vertices)	of which dense	share
k -cores, i.e. (1, 2)	880	8	0.9%
k -trusses, i.e. (2, 3)	120,401	23,624	19.6%
(3, 4)-nuclei	86,056	49,987	58.1%

Of the 880 subgraphs the k -core decomposition produces, eight are dense. The (3, 4)-nucleus decomposition produces fewer subgraphs than the k -truss decomposition, yet fifty thousand of them are dense, twice as many as the truss yields. The other graphs of Section 6 show the same pattern. One could continue to the (4, 5)-nucleus and beyond, and each step would pick out denser groups still, but each step also has to count larger cliques, and in real graphs the larger cliques are far more numerous (BlogCatalog has 51 M triangles and nearly twenty times as many four-cliques). Sariyüce et al. [13, 14] report that (3, 4) is the level that is most informative in practice, and it is the level this paper addresses.

Computing the (3, 4)-nucleus is expensive, in time and in space. It requires enumerating triangles, then four-cliques, then peeling the resulting clique hierarchy, and the four-cliques can outnumber the edges by hundreds of times. BlogCatalog has 2.1 M edges and 959 M four-cliques, and LiveJournal has 43 M edges and 9.9 billion, so holding the four-cliques, not the graph itself, is what sets the limit. This cost makes the problem a natural candidate for GPU acceleration, yet no GPU algorithm for (3, 4)-nucleus decomposition exists. The reason is structural. The canonical peeling algorithm enumerates higher cliques by recursing on the input graph, and it threads dynamic, pointer-based book-keeping through that recursion. Both patterns map poorly to a GPU, whose performance depends on flat, coalesced memory access and a fixed kernel structure rather than deep recursion and per-thread allocation. On the CPU, the parallel state of the art is ARB-NUCLEUS-DECOMP (ARB) [16], which enumerates cliques in parallel and peels them in bulk-synchronous rounds over an arboricity-oriented graph and parallel hash tables; it supersedes the earlier parallel local algorithms of Sariyüce et al. [12], and both are CPU algorithms. A recent counting-based method, CBND [20], avoids enumerating the s -cliques at all by computing supports through a clique-path index, again on a CPU. On the GPU the nucleus family has been taken no further than the k -truss, which sidesteps the recursion by needing a single level

of support [1, 3]. These truss-specific designs do not generalise to the multi-level structure of nucleus decomposition.

Our central idea is a *clique-reduction* primitive that removes the recursion. Given a graph and a vertex set, the clique reduction produces a new, linearised graph on which enumerating the next clique level reduces to ordinary triangle enumeration. Applying the primitive turns the recursive r -clique search of the canonical algorithm into a fixed sequence of triangle-enumeration passes over flat, array-based graphs. This is exactly the shape a GPU rewards. The resulting kernels are coalesced, free of dynamic in-kernel allocation, and light on locking. The primitive comes with provable linearisability¹ and composability properties that we use to establish correctness. For the peeling phase, where the local-update model iterates an h -index to a fixpoint to avoid contended updates [12], we instead let the GPU's cheap atomics absorb that contention and peel directly (Section 5).

The clique reduction also changes how memory is spent. The peel streams a flat support array rather than probing a hash table, touching four to six times fewer bytes per triangle than ARB. More importantly, the four-cliques need never be stored. Every four-clique that contains a triangle can be recovered from the triangles that share its three edges, so an index linear in the number of triangles lets each triangle's four-cliques be re-derived on demand, once to count support and again as each triangle is peeled. Where CBND avoids enumeration with an index on a CPU, we enumerate repeatedly and store no four-cliques, because on a GPU compute is abundant and memory is scarce. That is what takes the algorithm past the limit of device memory.

We summarise our contributions as follows.

- A clique-reduction primitive that recasts r -clique enumeration as triangle enumeration on a linearised, flat-array graph, together with the linearisability, composability and partitionability properties that make it correct and GPU-friendly.
- The first GPU algorithm for (3, 4)-nucleus decomposition, built from the primitive as a sequence of coalesced, allocation-free kernels, with a materialising mode for when the four-cliques fit in device memory and an on-demand mode that recomputes them from an index linear in the triangles when they do not.
- An empirical study against ARB at its best thread count and the recent CBND. Our algorithm on one H100 is 3 to 39× faster than ARB where the four-cliques fit, and on LiveJournal and Orkut, with 9.9 and 3.2 billion four-cliques, it finishes in under a minute, 2.5 and 4.8× faster than ARB, where CBND does not finish within an hour.

2 Related Work

Nucleus decomposition. Nucleus decomposition was introduced by Sariyüce et al. [13, 14], who generalised the k -core and k -truss to the (r, s) -nucleus and gave the peeling-based hierarchy. Their later local algorithms compute nucleus numbers without global peeling [12], and the framework has since been carried to probabilistic graphs [6]. ARB (Shi et al. [16]) peels in bulk-synchronous rounds over an arboricity-oriented graph and multi-level hash tables; it is the state of the art among parallel implementations, and our baseline. The same group later parallelised the construction of the full nucleus *hierarchy* (the containment tree over all k),

together with an approximate decomposition [17]; we compute exact nucleus numbers, the input to their two-phase hierarchy algorithm. Concurrent with our work, CBND (Zhang et al. [20]) reduces the work of nucleus decomposition by counting s -cliques through a clique-path index instead of enumerating them; its (3, 4) path is single-threaded, and Section 6 runs it as a second baseline. None of this work targets a GPU.

Cohesive subgraphs and clique listing. The k -core and k -truss are the most widely used cohesive-subgraph models and are exactly the (1, 2)- and (2, 3)-nucleus. The k -core is computed by degeneracy peeling [10]; the k -truss peels edges by triangle support, with efficient shared-memory [19] and triangle-graph [11] formulations. Both enumeration stages of our algorithm rest on clique listing; Chiba and Nishizeki gave the seminal arboricity-bounded algorithm [4], and modern parallel variants list k -cliques in sparse graphs efficiently [5] and count and peel them at scale [15]. Clique reduction reuses this machinery but recasts each level as triangle enumeration on a flat array, which is what makes it data-parallel.

GPU graph mining. Triangle counting [7] and k -clique counting [2] have efficient GPU implementations, but on the GPU the nucleus family has been taken no further than the k -truss [1, 3]. These truss designs exploit a single level of edge support, which the nucleus hierarchy does not reduce to. GPU correctness also turns on the memory model, where relaxed, non-coherent caches are a known hazard [18]; we therefore make every shared update with a device atomic, which needs no locks. To our knowledge, no prior GPU algorithm computes the (3, 4)-nucleus.

3 Preliminaries

We adopt the nucleus-decomposition framework of Sariyüce et al. [13, 14], following the formulation of Esfahani et al. [6].

Let $G = (V, E)$ be a simple, undirected graph on $n = |V|$ vertices and $m = |E|$ edges, where $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. We write $N(v) = \{u : \{u, v\} \in E\}$ for the neighbourhood of a vertex v and $\delta_v = |N(v)|$ for its degree. For an integer $k \geq 1$, a k -clique of G is a vertex subset $S \subseteq V$ with $|S| = k$ in which every pair of vertices is adjacent; $\times_k(G)$ denotes the set of all k -cliques of G . Fix integers $r < s$ and write R for a generic r -clique.

We orient G into a directed acyclic graph by ranking the vertices and directing each edge from its lower-ranked to its higher-ranked endpoint. Vertices are ranked by non-decreasing degree, with ties broken by vertex id; the relations $u < v$ and $\max(S)$ used below refer to this rank, with $\max(\emptyset)$ ranking below every vertex. This *degree orientation* allows each clique to be listed exactly once and bounds the clique-listing work by the arboricity of G [4]. We store the oriented graph in *compressed sparse row* (CSR) form, that is, one array holding the out-neighbours of every vertex in increasing rank order, concatenated in vertex order, and an offset array giving where each vertex's slice begins.

Definition 3.1 (Support). The s -support $\sigma_G(R)$ of an r -clique R is the number of s -cliques of G that contain R .

Definition 3.2 (s -connectedness). Two r -cliques R and R' are s -connected in G if there is a sequence $R = R_1, R_2, \dots, R_t = R'$ of r -cliques of G such that, for every i , the union $R_i \cup R_{i+1}$ is contained in some s -clique of G .

Definition 3.3 (k -(r, s)-nucleus and (r, s) -nucleus decomposition). For an integer $k \geq 0$, a k -(r, s)-nucleus is a maximal subgraph $H \subseteq G$ such that (i) $\sigma_H(R) \geq k$ for every r -clique R of H , and (ii) every pair of r -cliques of H is s -connected within H .

¹We use *linearisability* in the data-layout sense, namely that a clique-reduced graph flattens into linear arrays; it is unrelated to the Herlihy and Wing concurrency condition of the same name.

The (r, s) -nucleus decomposition of G is the collection of all its k -(r, s)-nuclei over all k ; it forms a hierarchy of progressively denser regions, and computing it means assigning every r -clique the largest k for which it lies in some k -(r, s)-nucleus.

The framework subsumes two classical cohesive-subgraph notions as special cases. The k -core is the $(1, 2)$ -nucleus (vertices supported by edges) and the k -truss is the $(2, 3)$ -nucleus (edges supported by triangles) [13]. This paper targets the $(3, 4)$ -nucleus, where $r = 3$ and $s = 4$. Triangles are the r -cliques to be peeled, and four-cliques provide their support. The *coreness* of a triangle is the largest k for which it lies in some k -($3, 4$)-nucleus. The standard way to compute corenesses is *peeling*: repeatedly remove an r -clique of minimum support k , record k as its coreness, and for each s -clique that contained it, decrement the support of the other r -cliques in that s -clique, never below k . Peeling is sequential in its dependence structure, since each removal changes the supports the next step reads, though the parallel versions of Section 5 remove all minimum-support cliques of a level in one round.

4 Clique Reduction

Computing the $(3, 4)$ -nucleus decomposition (Definition 3.3) needs, for every triangle, the number of four-cliques that contain it. The direct route enumerates the triangles of G , then the four-cliques, then peels triangles in increasing order of four-clique support. We unify the two enumeration stages with a single primitive, *clique reduction*, that lifts a graph one level so that finding the next clique size always reduces to triangle enumeration on a smaller, flat-array graph. Applied to G it yields the triangles; applied again it yields the four-cliques, and one GPU kernel (Section 5) serves both passes.

We assume $G = (V, E)$ has been oriented as in Section 3, so every edge runs from a lower-ranked vertex to a higher-ranked one and each clique is listed exactly once.

Definition 4.1 (Clique Reduction). Given an oriented graph $G = (V, E)$ and a vertex set $S \subseteq V$, the *clique reduction* of G with respect to S is the graph $G_S = (V_S, E_S)$ with

$$V_S = \{v \in V : v > \max(S)\},$$

$$E_S = \{(u, v) \in V_S \times V_S : u < v, S \cup \{u, v\} \in \times_{|S|+2}(G)\}.$$

Each edge $(u, v) \in E_S$ thus witnesses a clique $S \cup \{u, v\}$ of G . Figure 1 shows these reductions on a seven-vertex graph, where the single-vertex reductions $G_{\{0\}}, \dots, G_{\{4\}}$ expose the triangles of G and the pair reductions the four-cliques. Reducing by the empty set leaves the graph unchanged ($G_\emptyset = G$), so the input is its own level-zero reduction; the property that anchors the GPU realisation is the resulting layout.

Property 1 (Linearisability). Every clique-reduced graph G_S admits a flat two-array layout: an edge array of length $|E_S|$ holding the destination v of each $(u, v) \in E_S$, and a degree array of length $|V_S|$ whose prefix sum is the offset array used to index the edge array. For $G_\emptyset$ this layout is exactly the CSR adjacency list of Section 3.

Property 1 is the bridge to the rest of the paper. Every level is itself a flat edge, degree, and offset graph, so the GPU layout of Section 5 applies uniformly at each clique size. Moving up a level changes only the size of the tuple stored per edge (one vertex id at the input, three edge indices at the triangle level), not the arrays or the access pattern, which is what lets the same flat-array intersection serve every level.

THEOREM 4.2 (CORRECTNESS). Let $S \subseteq T \subseteq V$ with $T \setminus S \neq \emptyset$, T a clique of G and $\max(S) < \min(T \setminus S)$, and let $k \geq 1$. The $(|T| - |S| + k)$ -cliques of G_S that contain $T \setminus S$ are in bijection with the k -cliques of G_T ; the map removes $T \setminus S$.

PROOF SKETCH. Let C be a set of k vertices ranked above $\max(T)$. By Definition 4.1, every pair in $(T \setminus S) \cup C$ is an edge of G_S exactly when, together with S , it is a clique of G ; since T is a clique, this holds for all pairs precisely when $T \cup C$ is a clique of G . By the same definition, every pair in C is an edge of G_T exactly when, together with T , it is a clique of G , which again holds for all pairs precisely when $T \cup C$ is a clique of G . Both conditions are therefore equivalent to $T \cup C \in \times_{|T|+k}(G)$, and $C \mapsto (T \setminus S) \cup C$ is the bijection. The assumption $\max(S) < \min(T \setminus S)$ ensures $T \setminus S \subseteq V_S$, and the algorithm only ever applies the theorem in that situation, since each level reduces by one further vertex of an already-discovered clique. $\square$

The two passes of interest follow. For a vertex a , the edges of $G_{\{a\}}$ are exactly the triangles of G through a by Definition 4.1; and Theorem 4.2 with $S = \emptyset$, $T = \{a\}$, $k = 3$ makes the four-cliques of G through a the triangles of $G_{\{a\}}$. The next two corollaries let those passes be composed and split.

COROLLARY 4.3 (COMPOSABILITY). For $S, T \subseteq V$ with $\max(S) < \min(T)$ and $S \cup T$ a clique of G , $(G_S)_T = G_{S \cup T}$.

PROOF SKETCH. Both graphs have vertex set $\{v : v > \max(T)\}$. A pair (u, v) is an edge of $(G_S)_T$ when $T \cup \{u, v\}$ is a clique of G_S , that is, when every pair in $T \cup \{u, v\}$ forms a clique of G with S ; as $S \cup T$ is a clique, this is exactly $S \cup T \cup \{u, v\} \in \times_{|S|+|T|+2}(G)$, the condition for $(u, v) \in E_{S \cup T}$. $\square$

COROLLARY 4.4 (PARTITIONABILITY). Under the hypotheses of Corollary 4.3, $G_{S \cup T}$ can be computed from G_S alone, by intersecting, over the vertices of T , their neighbours ranked above $\max(T)$ and retaining the edges of G_S between the resulting common neighbours. Because the CSR slices are sorted by rank, each slice is touched only from its first neighbour ranked above $\max(T)$ onwards, a contiguous suffix. Hence for any $T_1, T_2, \dots$ each satisfying those hypotheses with the same S , the reductions $G_{S \cup T_i}$ all read the same G_S and none reads the output of another, so they can be computed concurrently, each producing its own output graph.

Corollary 4.3 justifies producing each clique level from the previous one rather than from G , and Corollary 4.4 makes the per-vertex reductions independent, with no reduction reading another's output, which is the parallelism the enumeration kernels of Section 5 exploit, and what lets an enumeration pass be split into independent segments when its scratch would exceed device memory. The subsequent peeling, by contrast, is global. The orientation of Section 3 does two things. Any total order gives each clique a unique lowest-ranked vertex, hence a single listing; ranking by non-decreasing degree bounds the listing work by the arboricity of G [4].

5 GPU Specialisation

We now specialise the clique reduction of Section 4 to a single GPU. The implementation follows three design principles. First, one thread per input element (a vertex, an edge, or a triangle, depending on the phase), so the device's wide single-instruction parallelism is used directly. Second, no allocation inside a kernel. Each phase pre-allocates a device buffer sized to a tight upper bound (for triangles, the number of length-two

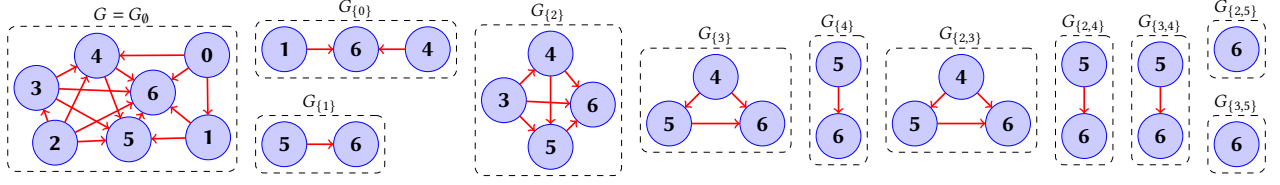


Figure 1: Running example: a seven-vertex graph $G = G_0$ and its successive clique reductions $G_{0}, G_{1}, \dots$ with respect to single vertices and pairs. Each reduction G_S retains only the edges that extend the clique on S , which is exactly the work one GPU thread performs when it intersects neighbour slices during enumeration.

paths $\sum_{(u,v) \in E} D[v]$) and threads claim output slots through one atomicAdd on a global counter. Third, all working structures are linearised and contiguous, which lets adjacent requests from a warp be combined rather than served separately (a *coalesced* access); this is the layout Property 1 guarantees at every level, so enumeration at every level is the same flat-array intersection.

Data layout. A graph of dimension DIM is stored as a templated struct `graph_t` with an edge array E , a degree array D , and an offset array O equal to the prefix sum of D . The template parameter DIM selects the clique dimension held in E : DIM=1 is an ordinary graph (one destination vertex per edge), DIM=2 is the triangle-level graph whose “edges” are triangles, and DIM=3 holds the four-cliques, each a tuple of four triangle indices, as a flat array the peel reaches through the incidence CSR built during four-clique extraction, rather than through an offset array of its own. Widening DIM changes only the *record*, the fixed-size tuple stored per edge, from one vertex id to three or four indices, and adds no pointer-based bookkeeping. Only the pair (E, D) is shipped to device memory; O is redundant given D and is rebuilt on the device by one parallel prefix scan, which shrinks the transfer.

Triangle enumeration. Triangle enumeration, a heavily optimised GPU primitive in its own right [7], is the first application of clique reduction and sets the template for every later level. For each oriented edge (u, v) we must inspect the neighbours of v , which begin in E at offset $O[v]$ and number $D[v]$. Gathering these per-edge ranges into one flat array is the role of the *multi-arrange* operation. Given two length- ℓ vectors b and c , multi-arrange emits in M the concatenation of the contiguous ranges $\lambda_{b[i], c[i]} = [b[i], b[i]+1, \dots, b[i]+c[i]-1]$ together with an offset vector N , where $N[i]$ marks the start of range i and $N[\ell] = |M|$. Feeding multi-arrange the per-edge offsets $O[E[\cdot]]$ as b and degrees $D[E[\cdot]]$ as c yields the indices in E of every second edge of a length-two path, with N partitioning M into the slice owned by each source edge. For example, take the oriented edge $(2, 3)$ of Figure 1: vertex 3 begins in E at $O[3] = 9$ and has $D[3] = 3$ neighbours, so this edge contributes $b = 9$ and $c = 3$, giving the slice $M = [9, 10, 11]$, the positions in E of 3’s neighbours $\{4, 5, 6\}$; intersecting them with 2’s own neighbour slice $\{3, 4, 5, 6\}$ matches 4, 5, and 6, three triangles. Triangle extraction is then a per-vertex merge. For each neighbour of a vertex we intersect the vertex’s own sorted neighbour slice of E with the multi-arranged two-hop slice, and emit a triangle on every match. Algorithm 1 states this kernel. The orientation of Section 3 is used in two places here. Every edge is inspected once, from its lower-ranked endpoint, and both slices being merged contain only vertices ranked above that endpoint, so each triangle is emitted exactly once, by its lowest-ranked vertex; and because the orientation is by degree, the aggregate merge work is bounded by the arboricity of G [4]. Each triangle is stored as a

Algorithm 1: Triangle extraction, one thread per vertex.

```

input : edge array  $E$ , offset array  $O$ , multi-arranged  $M$ ,  $N$ 
output: triangle array  $T$ , each entry a triple of edge indices
1  $u \leftarrow \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x}$ 
2 if  $u \geq |O| - 1$  then
3   return
4  $[f, l] \leftarrow [O[u], O[u+1]]$ 
5 for  $e \leftarrow f$  to  $l - 1$  do
6    $a \leftarrow f$ 
7    $p \leftarrow N[e]$ 
8   while  $a < l$  and  $p < N[e+1]$  do // two-pointer merge
9      $x \leftarrow E[a]$ 
10     $y \leftarrow E[M[p]]$ 
11    if  $x = y$  then
12       $t \leftarrow \text{warpAggregatedAdd}(\text{counter}, 1)$  // one atomic
13       $T[t] \leftarrow (a, e, M[p])$ 
14      if  $x \leq y$  then  $a \leftarrow a + 1$ 
15      if  $x \geq y$  then  $p \leftarrow p + 1$ 

```

triple of edge indices, the form the next level consumes directly. Note that M is laid out in source-edge order, so a thread walks M forward across its whole outer loop.

Four-clique extraction. By composability (Corollary 4.3), four-cliques are obtained by applying the same flat-array intersection to the triangle-level graph, whose triangles are the four-cliques of G , now merging three lists rather than two. As with GPU clique counting more broadly [2], the work stays within flat-array intersections. Each four-clique is recorded as four triangle indices into the triangle array, about sixteen bytes, and contributes one incidence to each of its four triangles, with no vertex tuples or hash keys retained. The (triangle, four-clique) incidences are then sorted by triangle on the device, which yields directly the structure peeling reads, a CSR $(O_{\text{INC}}, \text{INC})$ in which the slice $\text{INC}[O_{\text{INC}}[t]..O_{\text{INC}}[t+1]]$ lists the four-cliques on triangle t , next to the four-clique array QUD giving the four triangles of each. The support of t is the length of its slice, held in a flat array DD . On an instance that fits, the incidence CSR and the four-clique array stay on the device through peeling.

Peeling. Peeling determines the *coreness* (Section 3) of every triangle, the largest k for which it lies in some k -(3, 4)-nucleus. It proceeds level-synchronously and bucket-free. A CPU implementation keeps the triangles bucketed by current support and walks the buckets from the bottom up; maintaining such buckets under many concurrent updates is the kind of irregular bookkeeping a GPU handles poorly, so we drop them. Following parallel clique peeling [15], the device tracks two level variables, the current peel level `currentMin` and the next candidate level `nextMin`, and every thread tests its own triangle against `currentMin` directly. In each round (Algorithm 2) one thread owns one triangle. If the triangle is still unpeeled and its support is at or below `currentMin`,

Algorithm 2: GPU peeling, one thread per triangle.

input : incidence CSR O_{INC}, INC ; four-cliques QUD ; supports DD ; claim flags cl (init. 0); currentMin; nextMin = currentMin+1
output : coreness array $CORE$ (init. -1); nextMin

```

1  $t \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$ 
2 if  $t \geq |DD|$  or  $CORE[t] \neq -1$  or  $DD[t] > currentMin$  then return
3  $CORE[t] \leftarrow currentMin$ ;  $warpAggregatedAdd(peeled, 1)$ 
4 for  $i \leftarrow O_{INC}[t]$  to  $O_{INC}[t+1] - 1$  do
5    $q \leftarrow INC[i]$  // a four-clique containing  $t$ 
6   if  $atomicCAS(cl[q], 0, 1) \neq 0$  then continue
7   foreach  $w \in QUD[q]$  with  $w \neq t$  do
8      $old \leftarrow atomicSub(DD[w], 1)$ 
9     if  $old = currentMin + 1$  then nextMin  $\leftarrow currentMin$ 

```

the thread records its coreness and walks its incident four-cliques. A four-clique has four member triangles, and several can fall below the level in the same round. Whichever thread reaches it first *claims* the clique with an `atomicCAS` on a four-byte flag and the losers skip it, so exactly one claim succeeds per four-clique however many members die together, and the winner debits its other members with an `atomicSub`. This charges each destroyed four-clique exactly once and keeps the supports of unpeeled triangles exact, and it is why the peel needs no triangle-adjacency edge list, since the incidence CSR and the claim flags carry everything. When a decrement brings a member exactly down to the current level, the thread lowers nextMin so the next round revisits this level rather than advancing past it; the store is left plain, since every thread that takes this branch in a round writes the same value. A tiny update kernel then sets currentMin to nextMin and increments nextMin, and the driver loop repeats until every triangle is peeled. Recording each triangle's coreness as the level at which it is peeled is the sequential peeling rule, and the revisit keeps the levels non-decreasing, so the kernel computes the same (3, 4)-coreness as sequential peeling.

The local h -index iteration converges to the coreness without contended writes [12], but a GPU absorbs that contention in a single atomic instruction, so we peel directly, as ARB does.

Recomputing four-cliques on demand. Everything so far stores the four-cliques, and Section 6 shows what that buys while they fit. They do not always fit. Four-cliques grow far faster than edges: LiveJournal, with 42.9 M edges and 286 M triangles, has 9.93×10^9 four-cliques, and at the 36 bytes each one costs in the materialising pipeline (the clique, its four incidence entries, and its claim flag) that is 358 GB on an 80 GB card. Partitioning the enumeration does not help, since it bounds only the scratch space while the output remains the whole clique list. The observation that removes the list is that a four-clique never has to be written down, because it can be recovered from the triangles. Let triangle t have vertices $u < v < w$ and base edges (u, v) , (u, w) , (v, w) , the triple Algorithm 1 already emits. A four-clique containing t adds a vertex x adjacent to all three, so (u, v, x) , (u, w, x) and (v, w, x) must all be triangles, and x appears as the *third vertex* in the triangle lists of all three of t 's edges. We therefore build one edge-to-triangle CSR whose entries are keyed by third vertex, and a three-way merge over the lists of t 's three edges enumerates exactly the four-cliques on t , each match already carrying its three partner triangles' indices. This index holds three entries per triangle, not one per four-clique: 857 M entries for LiveJournal against 9.93×10^9 cliques, so memory becomes $O(\text{triangles})$ instead of $O(\text{four-cliques})$.

The same merge serves both passes. Support counting runs it once per triangle and keeps only the count. Peeling runs it again

for each dying triangle. A round first marks every live triangle at the current level as dying, then each dying triangle re-derives its four-cliques and debits the surviving partners. With no clique array there is no flag to claim, so ownership of a clique that loses several members at once is decided from the death records instead. The clique belongs to the first of its triangles to die, and among triangles dying in the same round to the lowest index, which a kernel boundary between the two phases makes readable. Each four-clique is re-enumerated once per member death, roughly four times the work of listing it once; that cost is paid in time, which the device has, rather than in memory, which it does not. The level also advances to the minimum surviving support rather than by one, since LiveJournal's supports are spread thinly over the range up to $k_{\max} = 351$ and empty rounds cost full passes.

Lock-light design. No phase takes a global lock. Outputs go to slots each thread reserves, so the result arrays carry no write conflicts, and the genuinely shared words (the reservation counter, the support array and the claim flags) are updated only through device atomics, which are correct under a GPU's relaxed memory model [18] and need no locks; nextMin is the sole plain store, and every thread that writes it in a round writes the same value, so the result does not depend on their order. The host launches kernels and exchanges scalar counters between phases but issues no within-phase control. Every structure the peel reads is device-resident. When an enumeration pass's scratch would exceed device memory it is split into segments instead, which bounds the scratch but not the output. The whole four-clique list must still be gathered on the host across the segments, uploaded, and used on the device to build the incidence CSR. Section 6 measures where host and device memory each run out.

6 Experiments

We evaluate the GPU algorithm against two CPU baselines. ARB is the parallel state of the art of Shi et al. [16], quoted at whichever thread count between 1 and 96 was fastest for each graph. CBND is the recent counting-based method of Zhang et al. [20], run from the authors' own code; its (3, 4) path is single-threaded, and we time it in its own favour by hoisting a clique-tree traversal out of its peel loop. Our algorithm runs on one NVIDIA H100 80 GB (HBM3), built with CUDA 13.2 and Thrust for `sm_90`; the CPU baselines run on a 96-core Intel Xeon 6972P processor, ARB built with g++ 12.3 at -O3 and its bundled parlay scheduler. Every time comes from the implementation's own algorithm timer and excludes the one-time load of the graph from disk (for ARB, also its `.adj` conversion). All three implementations agree on the four-clique count and on $k_{\max}$ wherever they complete. Datasets come from SNAP and the Laboratory for Web Algorithmics, with isolated vertices, self-loops and duplicate edges removed (Table 1). Friendster, whose 4.2×10^9 triangles would put the on-demand index alone at roughly 150 GB, is out of reach on a single device.

6.1 Wall-clock time

On the six graphs whose four-cliques fit in device memory, the materialising GPU algorithm of Section 5 outperforms ARB by 3.1 to 39.4 $\times$ (Table 1). The margin is widest on FourSquare at 39.4 $\times$ and narrowest on Slashdot0811 at 3.1 $\times$, the graph with the fewest four-cliques. On the two graphs whose four-cliques do not fit (the resident representation alone is 358 GB for LiveJournal and 116 GB for Orkut, against the card's 80, with a higher peak while the incidence is built), the on-demand mode is the only GPU method that runs; it finishes both in under a minute and

Table 1: Datasets, ordered by $|E|$, and wall-clock time in seconds. ARB at its best thread count of 1–96 on the Xeon processor; CBND single-threaded; ours on one H100, storing the four-cliques (mat.) or recomputing them on demand (o.d.). Gain is the best GPU time against ARB. OOM: materialising the four-cliques exceeds available host or device memory. $k_{\max}$ agrees across every implementation that completes.

Dataset	$ E $	four-cliques	$k_{\max}$	ARB	CBND	Ours		
						mat.	o.d.	Gain
Slashdot0811	469 K	2.0 M	26	0.18	5.51	0.058	0.089	3.1×
BlogCatalog	2.1 M	959.3 M	71	24.34	>1 h	6.06	9.81	4.0×
FourSquare	3.2 M	162.6 M	31	19.77	458.1	0.50	7.01	39.4×
amazon-2008	3.5 M	4.2 M	8	0.24	7.12	0.018	0.020	13.5×
digg	5.9 M	551.4 M	59	19.12	>1 h	3.26	8.00	5.9×
cit-patent	16.5 M	3.5 M	18	0.62	15.84	0.043	0.056	14.4×
LiveJournal	42.9 M	9.93×10^9	351	113.1	>1 h	OOM	45.7	2.5×
Orkut	117.2 M	3.22×10^9	65	177.2	>1 h	OOM	36.9	4.8×

outperforms ARB at its best thread count by 2.5× and 4.8×. Both materialising failures are measured, not projected: LiveJournal exhausts host memory, Orkut the device. CBND completes only on the four graphs with the fewest four-cliques, where it is 23 to 31× slower than ARB, and times out at one hour on the rest, including both large graphs; the largest graph in its own (3, 4) evaluation has 22.3 M edges [20]. We note that $s = 4$ is where a counting-based index has least to save, since the s -clique enumeration it avoids grows with s , as does CBND’s reported advantage over enumeration-based methods [20].

These are end-to-end comparisons of one device against ARB at its best thread count between 1 and 96, so the gains combine hardware, algorithm, implementation and layout, and we do not separate them. The device is taken as given; no (3, 4)-nucleus decomposition ran on a GPU before this work, and clique reduction is what makes its bandwidth reachable. ARB is itself 12.6 to 143.5× faster than the earlier parallel nucleus tool of Sariyüce et al. [13] on the same node, which is why we compare against it and not the older tool.

6.2 Storing four-cliques or recomputing them

The on-demand column of Table 1 prices the trade of Section 5. Where the four-cliques fit, recomputing them costs 1.1 to 2.5× on five of the six graphs and 14× on FourSquare, whose one vertex of degree 106,218 makes its intersection lists extremely uneven; even so the on-demand mode outperforms ARB on all six. The cost shows in the phases. On BlogCatalog it replaces 3.4 s of four-clique enumeration with 0.16 s of index construction and pays with a peel that rises from 2.4 to 9.5 s. Where the four-cliques do not fit, the trade is what makes the computation possible at all. The materialising representation costs 36 B per resident four-clique, but the sort that builds the incidence holds the triangle keys alongside, taking the peak to at least 52 B before its own temporaries, so the ceiling sits well below what the resident figure alone would suggest. We report it as measured rather than derived. BlogCatalog, with 959.3 M four-cliques, is the largest instance that completes, and every degree-ordered prefix of Orkut fails building the incidence, the smallest of them already holding 1.22×10^9 . The on-demand index costs 36 B per triangle plus its edge-offset array, 10.6 GB for LiveJournal and 23.5 GB for Orkut, with a measured peak device footprint of 17 and 37 GB once the per-triangle working arrays are added. Its peel re-enumerates each four-clique about four times, exchanging extra work for the

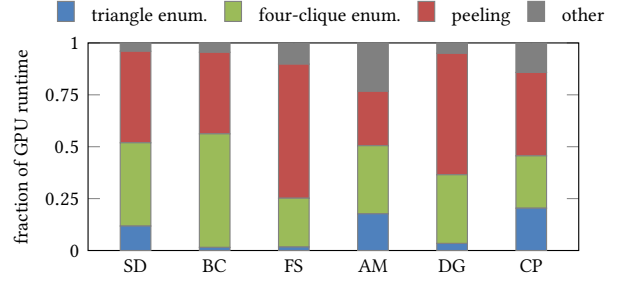


Figure 2: GPU time per phase in the materialising pipeline, as a fraction of runtime, for the six graphs of Table 1 whose four-cliques fit, in table order. “Other” is graph construction and host–device transfers.

smaller footprint. The work is paid in time, which the device has, rather than memory, which it does not.

Where the time goes. In the materialising pipeline (Figure 2), peeling and four-clique enumeration dominate, while triangle enumeration and the fixed cost of using a GPU at all matter only on the graphs that finish in tens of milliseconds. So clique reduction makes enumeration cheap enough to expose the peel as a cost in its own right, and on the densest graphs, where four-cliques outnumber triangles nearly twenty to one, materialising them is itself a substantial share, which is what the on-demand mode removes.

Memory per triangle. ARB keeps each triangle in a hash table keyed by its three vertices, 16 to 24 bytes with the support and open-addressing slack, and every support update is a probe. Our layout keeps the identity once (three edge indices, 12 bytes) and the support as a 4-byte entry indexed by triangle id, so the total is the same but the peel streams 4 contiguous bytes per element against 16 to 24 hashed ones, a working set four to six times smaller that follows from Property 1, not from any device.

7 Conclusion

We introduced clique reduction, a primitive that lifts a graph one clique order at a time with provable linearisability, composability and partitionability, and built on it the first GPU algorithm for (3, 4)-nucleus decomposition, in which enumeration and peeling operate over a flat edge, degree, and offset layout that maps directly onto device memory. Our algorithm on one H100 is 3 to 39× faster than ARB at its best thread count where the four-cliques fit in device memory, and where they do not, an on-demand mode that re-derives them from an index linear in the triangles decomposes LiveJournal’s 9.9 billion four-cliques in 46 seconds, on a graph that CBND, a recent CPU method, cannot finish within an hour.

Nothing in the primitive is specific to (3, 4). Clique reduction lifts any level to the next, so the (r, s) -nucleus for $s = r + 1$ is the same pipeline with $r - 1$ reductions before the peel, and the k -core and k -truss are its (1, 2) and (2, 3) instances with zero and one reduction; the flat layout and the peel apply unchanged at every level. The on-demand mode stretches a fixed device budget further, since its index stays linear in the $(s-1)$ -cliques rather than the s -cliques, while the peel re-enumerates each s -clique once per member death. That re-enumeration is the limitation, because its cost grows with s , so the trade of time for memory becomes steeper as the order rises.

Artifacts

The CUDA source for both modes of our algorithm, materialising and on-demand, is at <https://github.com/mexuaz/gpu-nucleus34>, where the README gives the build and run instructions. All datasets are public and listed in Table 1.

References

- [1] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S. Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on k-truss Decomposition on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2019.8916285
- [2] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2022. Parallel K-clique counting on GPUs. In *ICS '22: 2022 International Conference on Supercomputing*. ACM, 21:1–21:14. doi:10.1145/3524059.3532382
- [3] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *Proc. VLDB Endow.* 13, 10 (June 2020), 1751–1764. doi:10.14778/3401960.3401971
- [4] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223. doi:10.1137/0214017
- [5] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs*. In *Proceedings of the 2018 World Wide Web Conference (Lyon, France) (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 589–598. doi:10.1145/3178876.3186125
- [6] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. 2022. Nucleus Decomposition in Probabilistic Graphs: Hardness and Algorithms. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 218–231. doi:10.1109/ICDE53745.2022.00021
- [7] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *SIGMOD '21: Proceedings of the 2021 International Conference on Management of Data*. 736–748. doi:10.1145/3448016.3452815
- [8] Wensheng Luo, Chenhao Ma, Yixiang Fang, and Laks V. S. Lakshmanan. 2023. A Survey of Densest Subgraph Discovery on Large Graphs. arXiv:2306.07927 [cs.SI]
- [9] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V.S. Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient Algorithms for Densest Subgraph Discovery on Large Directed Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1051–1066. doi:10.1145/3318464.3389697
- [10] David W. Matula and Leland L. Beck. 1983. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM* 30, 3 (jul 1983), 417–427. doi:10.1145/2402.322385
- [11] Mohsen Rezvani and Mojtaba Rezvani. 2022. Truss decomposition using triangle graphs. *Soft Computing* 26, 1 (2022), 55–68. doi:10.1007/s00500-021-06468-9
- [12] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proc. VLDB Endow.* 12, 1 (Septemeber 2018), 43–56. doi:10.14778/3275536.3275540
- [13] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Umit V. Catalyurek. 2015. Finding the Hierarchy of Dense Subgraphs Using Nucleus Decompositions. In *Proceedings of the 24th International Conference on World Wide Web (Florence, Italy) (WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 927–937. doi:10.1145/2736277.2741640
- [14] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2017. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Transactions on the Web* 11 (7 2017). Issue 3. doi:10.1145/3057742
- [15] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel Clique Counting and Peeling Algorithms. arXiv:2002.10047 [cs.DS]
- [16] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Theoretically and Practically Efficient Parallel Nucleus Decomposition. *Proceedings of the VLDB Endowment* 15 (2021), 583. Issue 3. doi:10.14778/3494124.3494140
- [17] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2024. Parallel Algorithms for Hierarchical Nucleus Decomposition. *Proceedings of the ACM on Management of Data* 2, 1, Article 32 (2024), 27 pages. doi:10.1145/3639287
- [18] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 578–590. doi:10.1109/HPCA.2013.6522351
- [19] Shaden Smith, Xing Liu, Nesreen K. Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. 2017. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. doi:10.1109/HPEC.2017.8091049
- [20] Wenqian Zhang, Zhengyi Yang, Dong Wen, Yi Ding, Wenjie Zhang, and Xuemin Lin. 2026. Nucleus Decomposition Revisited: An Efficient Counting-Based Approach. *Proceedings of the ACM on Management of Data* 4, 3, Article 215 (2026), 26 pages. doi:10.1145/3802092