ABSTRACT

k-Core decomposition is a well-studied community detection problem in graph analytics in which each k-core of vertices induces a subgraph where all vertices have degree at least k. The decomposition is expensive to compute on large graphs and efforts to apply massive parallelism have had limited success. This paper presents a vectorisation of the problem that reframes it as a composition of vector primitives on flat, 1d arrays. With such a formulation, we can deploy highly optimised Deep Learning GPU and SIMD frameworks. On a moderate GPU, using PyTorch, we obtain up to 8x improvement over the best parallel state-of-the-art implemented in C++ and running on an expensive 32-core machine. More importantly, our approach represents a novel abstraction showing that redesigning graph operations as a series of vectorised primitives makes highly-parallel analytics both easier and more accessible for developers. We posit that such an approach can vastly accelerate the use of cheap GPU hardware in complex graph analytics.

KEYWORDS

graph analytics, k-core decomposition, parallel algorithms, vectorization, GPGPU, SIMD, PyTorch

1 INTRODUCTION

Abstractions simplify the complex. They make experienced developers more efficient and allow novice developers to do things they otherwise could not. This paper abstracts graph analytics on GPUs. Similar to how [6] and [11] reframed shortest paths and reachability queries in terms of the declarative operators well known to relational databases, we reframe k-core decomposition in terms of the vector primitives well known to modern Deep Learning frameworks like PyTorch and Tensorflow. We focus on a high-level declaration of how wide, data-level parallelism can be exposed, leaving the mechanics of parallelisation to highly-optimised libraries.

Graph analytics, in general, is notoriously difficult to parallelise effectively because of the pointer-chasing, unpredictable, irregular access patterns and poor thread saturation. k-Core decomposition [9] is a well-studied graph analytics problem that we describe in Section 2 and illustrate in Figure 1. It is particularly difficult to parallelise because work-efficient solutions incur substantial synchronisation. We ‘vectorise’ the algorithm (in the ML sense); that is to say, we show how to re-express the sequential state-of-the-art [2, 9] by composing vector operations on flat 1d arrays.

The only existing GPU algorithm [12] is outperformed by pre-existing multi-core algorithms [3, 4, 7], based on numbers reported in each paper. Compared to the multi-core algorithms, we obtain 4–8x speed-up on real data using an approach that is both easier and more accessible. In summary, our contributions are as follows.

(1) We provide a compelling example for re-framing graph analytics in terms of Deep Learning frameworks.
(2) We present an efficient expression of a new vector primitive, namely retrieving all the neighbors of a set of vertices, which is of independent interest for other graph analytics problems.
(3) We improve upon the k-core decomposition state-of-the-art by 4–8x, using more readily available and cost-efficient hardware (a GPU versus a 32-core server).

2 BACKGROUND

We represent networks using undirected graphs. We denote an undirected graph by \( G = (V, E) \), where \( V \) is the set of vertices, and \( E \) is the set of edges. We set \( n \) and \( m \) to be \( |V| \) and \( |E| \), respectively. Given a vertex \( v \), we denote by \( d_G(v) \) the degree of \( v \). We set \( d_{\text{max}}(G) = \max\{d_G(v) : v \in V\} \).

Let \( K \subseteq V \) be a subset of vertices of a graph \( G = (V, E) \). Graph \( G(K) = (K, E_K) \), where \( E_K = \{(u, v) \in E : u, v \in K\} \) is called the subgraph of \( G \) induced by \( K \).

Now, \( G(K) \) is a k-core iff the following conditions are true: (Degree) for each \( v \in K \), \( d_{G(K)}(v) \geq k \), and (Maximality) for each \( K' \), such that \( K \subset K' \) there exists \( u \in K' \setminus K \), such that \( d_{G(K')}(u) < k \).

From the maximality condition it follows that for each \( k = 1, 2, \ldots, d_{\text{max}}(G) \), there exists exactly one k-core in \( G \) (which could possibly be empty). Given \( k \in [1, d_{\text{max}}(G)] \), we denote the k-core of \( G \) by \( C_k(G) \). Finally, we have that a vertex \( v \in G \) has coreness \( k \), denoted \( c(v) \), if and only if it is a vertex in graph \( C_k(G) \).

Example Figure 1 gives a graph where vertices are coloured by their coreness. The 3-core is given by \( v_1 \sim v_3 \sim v_6 \); the 2-core, vertices \( v_5 \sim v_6 \) and the 1-core, \( v_7 \sim v_{14} \). All cores > 3 are empty. \( v_7 \) illustrates why the problem is challenging: although it has degree 3 and neighbours two degree-2+ vertices, the 1-coreness of \( v_{14} \) cascades all the way to \( v_7 \), leaving \( v_7 \) with only one neighbour in the 2-core.

3 EXISTING APPROACHES

There are two general approaches to k-core decomposition: peeling algorithms (based on [2, 9]) and vertex-centric algorithms [10].
Peeling Graph peeling is an $O(m)$ process for computing cores. At a high level, it works as follows:

1. Set the coreness of every vertex to equal its degree
2. Iteratively delete the vertex $v$ of smallest coreness $c(v)$.
3. Decrement $c(u)$ for each neighbour $u$ of $v$ s.t. $c(u) > c(v)$.

The $O(m)$ asymptotic efficiency comes from only visiting each edge a constant number of times and by using an efficient, dynamic data structure to maintain the list of vertices in ascending order of coreness. Given the small domain of possible coreness values, this sort order can be maintained by combining a preliminary bin sort by coreness with the insight that vertices can move at most one bin per iteration of the algorithm and only once per incident edge [2].

Vertex-Centric The vertex-centric approach [10] was designed for shared-nothing architectures. At a high level, it works as follows:

1. Each vertex $v$ obtains coreness estimates from its neighbours.
2. The estimate of $c(v)$ is updated per the neighbour estimates.
3. The process is iterated until convergence.

This approach reduces communication and scales out well, but has a cost linear in $n$ per iteration. It has been shown to be slower than peeling approaches in a shared-memory, single-core setting [8]; we confirm this also for multi-core architectures (c.f., Section 5).

Shared-Memory Parallel Algorithms State-of-the-art multi-core approaches [3, 4, 7] have followed the peeling paradigm. They expose parallelism in recognising that multiple vertices typically tie for the smallest coreness value. They can all be concurrently deleted and the coreness of their neighbours can be concurrently decremented (perhaps multiple times). The algorithm implemented in the Julienne software [4] (revisited in [5]) focuses on maintaining asymptotic work-efficiency by introducing methods to generalise the bin-sorted data structure to multiple threads. The PKC [7] algorithm improves memory performance by reconstructing the graph in parallel once a large percentage of the vertices have been deleted. The Park [3] algorithm exposes more parallelism, but by means of increasing the (sequential) asymptotic complexity to $O(m + n k_{max})$, where $k_{max}$ denotes the largest coreness value in the graph.

GPUs There is very little work on GPUs for $k$-core decomposition [12]. This is unsurprising given that graph peeling does not expose enough data-level parallelism even to saturate a multi-core machine (c.f., Section 5). In contrast, GPU research has focused on different problems that are more compute- and memory-intensive, such as temporal core [15] and k-truss [1, 14] decompositions.

The algorithm in [12] proceeds with two directions of peeling. At each iteration, it uses peeling to determine the maximum coreness in the graph, $k_{max}$; then, it removes that $k$-core, i.e., it peels from highest coreness to lowest. This exposes more parallelism and shrinks the graph faster, by focusing first on the highest-degree vertices, but at the cost of computing a $k_{max}$ value $k_{max}$ times. On a common moderate dataset we use, soc-LiveJournal1, the runtime that [12] reports (60.755 sec) is orders of magnitude higher than our runtime (102 ms) using an identical GPU processor (c.f., Section 5).

### Algorithm 1: Our vectorized algorithm.

```plaintext
input : $I, D, \Pi$
output: $k$
1. $k = 1$ // Processing $k^{th}$ coreness
2. $N = I(n)$ // Numerical range from 0 to $n - 1$
3. $CD = D$ // Immutable copy of array $D$
4. $B = N[D[N] <= k]$ // Vertices with degree $\leq K$
while $N.size() > 0$
do
5. $D[B] = 0$
6. $K[B] = k$
7. $J = I[I[B] \odot CD[B]]$
8. $H, T = \text{unique}(J[D[J] > 0])$
9. $D[H] = T$
10. $B = H[D[H] <= k]$
on
11. $k++$
12. $N = N[D[N] >= k]$
13. $B = N[D[N] == k]$
```

### Algorithm 2: The multi-range operation $M = S \odot C$.

```plaintext
input : $S, C$
output: $M$
1. $R = \text{zeros}(\text{size}(S))$
2. $R[1:] = \text{cumsum}(C)[:-1]$
3. $T = \text{ones}(\text{sum}(C))$
4. $T[R] = S$
5. $T[R[1:]] += 1 - (S[0] + C[1-1])$
6. $M = \text{cumsum}(T)$
```

## 4 PROPOSED APPROACH

We follow the peeling paradigm, but take a different approach to parallelising it. We reframe the problem in terms of vector primitives to maximally expose SIMD parallelism, then compose these primitives using highly optimised GPU vector processing libraries.

Algorithm 1 describes the overall procedure, following the “fancy” and “boolean” array notation of advanced array indexing in Numpy and the Torch libraries of Python [13]. Function `unique()` applies to a pre-sorted array and returns two arrays representing unique values and their frequency.

**Input** The algorithm requires three vectors as input, which together form a flattened adjacency list. Vector $I$ is of length $m$ and contains the destination of each edge, sorted by origin. Vector $D$ is of length $n$ and contains the degree of each vertex. Finally, $\Pi$ is also of length $n$ and provides the index in $I$ where each vertex’s neighbours begin.

**Determining Coreness** The procedure begins at coreness $k = 1$ by initialising a sorted array $N$ of all vertex ids and loops until $N$ is empty, iteratively peeling away cores. At each iteration, we obtain the set of vertices, $B$, whose degree does not exceed the current coreness value (Line 4, 12 & 15). All vertices in $B$ are assigned coreness of $k$ (Line 8) and are deleted (by zeroing their degree in $D$ (Line 7)). Lines 9–11 retrieve the neighbours of vertices in $B$ with
nonzero degree (including duplicates) and decreases their degree by the number of neighbors they have in $B$. We increment $k$ and update $N$ whenever $B$ is empty.

**multi-arange Operation** An essential part of Algorithm 1 is the *multi-arange* operation, $\diamond$, called in Line 9. The *multi-arange* operation is not defined directly in the Numpy or Torch libraries. Here we show how to define it in terms of optimized vector primitives. multi-arange is a binary operation that transforms two equal-length vectors, $S$ and $C$, into an output vector of length $\sum_{c \in C} S$. $S$ denotes a set of start indices and $C$ denotes a set of counts. For each $(s_i, c_i) \in (S, C)$, it generates the series $s_i, s_i + 1, \ldots, s_i + c_i - 1$. For example, $[2, 4, 1] \diamond [2, 1, 3] = [2, 3, 4, 1, 2, 3]$. This function is used in the main algorithm to quickly generate the indices in array $I$ of the neighbors of a selected set of vertices based on their start positions stored in vector $U$ and their degree.

Algorithm 2 describes it in terms of vector primitives. The function accepts two vectors of same length, $S$ and $C$. It uses the `cumsum()` function, which returns the cumulative sum of all the elements of the input vector and has a linear order. A naive alternative solution for this function is to use two nested loops over arrays $S$ and $C$ and generate the indices in subsequent memory locations. But the `cumsum()` function introduced in the Torch library is a parallel vector operation and using that along with other primitive vector operations like simple arithmetic and assignment results into a fully parallel vector function as a whole with linear order.

Our solution for this function is initializing a vector of output size in a way that applying a `cumsum()` function will produce the desired values. As the output vector consists of multiple segments of independent series, we require some reset values along the output vector before applying `cumsum()`. The indices for these reset values are stored in vector $R$. The values of the vectors for the above example $[2, 4, 1] \diamond [2, 1, 3]$ after each line are follows. (1) $R = [0, 0, 0]$, (2) $R = [0, 2, 3]$, (3) $T = [1, 1, 1, 1, 1]$, (4) $T = [2, 1, 4, 1, 1, 1]$, (5) $T = [2, 1, 11, 31, 1]$, (6) $M = [2, 3, 4, 1, 2, 3]$. 

### 5 EXPERIMENTS

#### 5.1 Experimental setup

**Implementations** We compared the vectorized algorithm (Vectr) with four state of the art algorithms designed for multi-core CPU architectures: Park [3], PKC and PKC-o [7] and the vertex-centric MPM [4]. We also implement the sequential state-of-the-art, BZ [2].

We implemented Vectr on a CUDA 8.0 platform with Torch library version 2017.1030 using Python 3.7. We implemented the multi-core algorithms in C++, starting from C implementations made publicly available by [7]. We use OpenMP 2.1.1 and compile with the Intel 2016.4 compiler, using the -O3 optimisation level. Our source code is publicly available.$^3$

**Hardware** We use a Tesla P100-PCIE-12GB GPU and a dual-16 core, Intel® Xeon® CPU E5-2683 v4 @ 2.10GHz (Broadwell microarchitecture) CPU with hyper-threading disabled, i.e., 32 cores in total. The OS was a CentOS Linux release of 7.5.1804 (Core).

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Nodes</th>
<th>Edges</th>
<th>Size (MB)</th>
<th>$d_{avg}$</th>
<th>$d_{max}$</th>
<th>$k_{avg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>403k</td>
<td>4.9M</td>
<td>37</td>
<td>7</td>
<td>1,076</td>
<td>7.2</td>
</tr>
<tr>
<td>H09</td>
<td>4.8M</td>
<td>85.7M</td>
<td>658</td>
<td>15</td>
<td>19,409</td>
<td>9.4</td>
</tr>
<tr>
<td>H11</td>
<td>1.1M</td>
<td>112.8M</td>
<td>860</td>
<td>100</td>
<td>11,468</td>
<td>60.0</td>
</tr>
<tr>
<td>LJ</td>
<td>2.2M</td>
<td>229.0M</td>
<td>1747</td>
<td>105</td>
<td>13,107</td>
<td>61.0</td>
</tr>
</tbody>
</table>

Table 1: Statistical properties of the datasets

**Datasets** The datasets are real data and taken from the Laboratory for Web Algorithmics$^2$ and Stanford SNAP collection.$^3$ Isolated vertices have been removed and the directed graph (LJ) was transformed to be undirected. Table 1 summarises the statistical properties of the datasets.

*amazon-2008* (AM) is a dataset of books, where a bidirectional edge between books indicates that they are similar. *soc-LiveJournal1* (LJ) is a social network, where a directed edge $(u, v)$ indicates that user $u$ has befriended user $v$. *hollywood-2009* (H09) is a social graph, captured in 2009, in which an edge indicates that two actors/actresses have co-appeared in a movie. Finally, *hollywood-2011* (H11) is the evolved actors/actresses graph in 2011.

#### 5.2 Results and Discussion

Table 2 shows the raw execution times for each algorithm on each dataset and each applicable core count, starting from the point that the graph has been loaded into memory and a common, flattened adjacency list has been created. Vectr includes the PCle3 transfer to the GPU device.

**Baselines and Fairness** Observe in Table 2 that PKC is consistently within 2x of BZ, indicating a small and easily amortizable parallel overhead, despite the extra term in the asymptotic complexity. Figure 2 illustrates the parallel speed-up obtained by each algorithm on each dataset. AM is very small, so all algorithms struggle to expose enough parallelism for 32 cores. Despite having 32 physical cores on 2 sockets, no multi-core algorithm consistently obtains more than an 8x speed-up. ParK often gets better parallel scalability than PKC, but the parallel speed-up of PKC is competitive and, additionally, PKC has better single-threaded performance; thus, it is still faster on 32 cores.

**Relative Performance** Figure 3 illustrates the performance of each algorithm, relative to PKC. A y-value of 1.0 indicates parity with PKC; a value larger (or smaller) than 1.0 indicates better (or worse) performance relative to PKC.

$^2$[tosca.ifi.unicamp.br/datasets.php](https://tosca.ifi.unicamp.br/datasets.php)

$^3$[https://snap.stanford.edu](https://snap.stanford.edu)

![Figure 2: Speedup of baselines on 32 cores versus 1 core.](image-url)
We can see from the experiments that vectorizing algorithms could achieve considerable speed up by doing our experiments on GPU accelerated machines. Besides, in recent years, many libraries introduced APIs for basic operations on large arrays like Torch, Octave, OpenAcc and Thrust. Employing these APIs could save the algorithm designers from the hassle of developing parallel programs while benefiting from it. Most of these libraries adopt a unique syntax for different multi-core hardware platforms ranging from CPU and GPU to DSP. So it allows the developer to experiment with the algorithm on different platforms without necessarily applying any changes to the source code.

ACKNOWLEDGMENTS

This research was enabled in part by support provided by the WestGrid organization (https://www.westgrid.ca) and by Compute Canada (https://www.computecanada.ca).

REFERENCES


Figure 3: Speedup of algorithms versus PKC as a baseline. Vectr uses the GPU; all others were run on 32 physical cores.

Table 2: Execution times (ms) for algorithms over selected datasets, using 1 (1t) and 32 cores (32t)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BZ 1t</th>
<th>MPM 1t</th>
<th>PKC 32t</th>
<th>PKC-o 32t</th>
<th>ParK 1t</th>
<th>Vectr 32t</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon-2008</td>
<td>354</td>
<td>1411</td>
<td>151</td>
<td>190</td>
<td>24</td>
<td>191</td>
<td>23</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>6953</td>
<td>34272</td>
<td>8911</td>
<td>5198</td>
<td>864</td>
<td>10193</td>
<td>1113</td>
</tr>
<tr>
<td>hollywood-2009</td>
<td>5555</td>
<td>36405</td>
<td>8394</td>
<td>4844</td>
<td>1113</td>
<td>11532</td>
<td>1440</td>
</tr>
<tr>
<td>hollywood-2011</td>
<td>11936</td>
<td>96203</td>
<td>27433</td>
<td>9787</td>
<td>2162</td>
<td>14852</td>
<td>2535</td>
</tr>
<tr>
<td>speed-up</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2: Execution times (ms) for algorithms over selected datasets, using 1 (1t) and 32 cores (32t)