The Matrix Exponential as a Graph Centrality Measure: A Parallel Lanczos Method Using CUDA

Hugh Delaney

September 16, 2021

Supervisors: Kirk M. Soodhalter, Jose Refojo
Course: M.Sc in High Performance Computing
University: Trinity College Dublin

Abstract

Networks are ubiquitous structures, appearing in virtually all scientific or non-scientific fields of study [1]. Networks may represent the interactions between animals in an ecosystem, links between roads in a network, academics citing other academics, users interacting in a social network, as well as countless other representations.

In the analysis of networks it is of great importance to identify central and non-central nodes. There are countless ways of measuring the centrality of nodes in a network, all with slightly different interpretations of what it means to be *central*. Finding highly central and poorly central nodes is essential in maintaining, managing, and living in real life networks. A power grid needs to identify poorly connected nodes in order to prevent outages. An airline needs to identify and avoid highly central, or oversubscribed air routes. An academic might shoot to fame if they can persuade a highly central researcher to cite them. The list goes on.

Matrix functions are a vital tool for computing node centrality in networks and graphs. The most commonly used matrix function is the matrix exponential, whose calculation provides particular insight into the structure of networks. The unique lens that the matrix exponential offers to analyze graphs will be discussed in section 1.4.

The aim of this project is to compute the centrality of nodes in an undirected graph ($A^T = A$). The action of the exponential matrix function $f(A)x = e^A x$ will be used as a measure of node centrality. The desired quantity $e^A x$ will be approximated without forming e^A explicitly, instead using Krylov subspace methods, most notably the Lanczos method. This makes computing $e^A x$ scalable to the limits of the memory hardware, which will be tested in this project.

The implementation will parallelize all possible operations with CUDA, and chart speedups as well as accuracy. A dual-card CUDA Lanczos method will also be trialled.

Acknowledgements

Some nodes and networks I would like to thank:

- The lecturers and tutors of the MSc program. Their hard work and dedication has given me a lifelong passion for computing and performance. Special thanks to Richie Morrin, who tirelessly answered every stupid question I fielded to him, and taught me almost everything I currently know about computers.
- My supervisor Kirk Soodhalter, who has helped me reacquaint myself with the joy of maths. I was grateful to have the opportunity to brainstorm with him in developing this project. Among the many things he has taught me is that maths should be a social activity.
- My secondary supervisor José, who, during the CUDA module, responded to on average about three emails a day from me. I think I finally understand it now! I enjoyed the nerdy chats and the thrill of getting code to run faster in new and unusual ways.
- My wife Reem. It is only with your constant support that I am able to achieve anything. The last year or so of working at home has been a joy with you.
- Mum, Dad, Milo and Ronan. During a strange year you are the network that has been the most important to me. Thank you for everything. I look forward to being all together at Christmas to have a nice big argument.
- All of the aunts, uncles, and cousins in my life. I am incredibly grateful for you all, and hope to see more of you in the coming year.
- My classmates in the MSc, all of whom I have only met in person a once or twice. I look forward to going to the pub with you all soon!

Contents

Notes

- The words graph and network shall be used interchangeably.
- The words node and vertex shall be used interchangeably.
- G will refer to a graph.
- V will refer to the set of vertices of a graph.
- E will refer to the set of edges of a graph.
- A will refer to a graph's adjacency matrix.
- n will refer to the number of vertices in a graph, which is equivalent to the dimension of said graph's adjacency matrix.
- A will be referred to as *sparse* if the number of nonzeros in the matrix is far less than n^2.
- r will refer to the dimension of the Krylov subspace with respect to a given matrix A and vector b, $\mathcal{K}_r(A, b) = \text{span}\{b, Ab, \ldots, A^{r-1}b\}$
- a_{ij} or $[A]_{ij}$ will refer to the entry in the i^{th} row and j^{th} column of the matrix A.
- b_i will refer to the i^{th} column of the matrix B, or the i^{th} entry of a column vector b.
- $I \in \mathbb{R}^n$ will refer to the identity matrix

$$i_{jk} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

- A^T will refer to the transpose of A, the reflection of the matrix across its diagonal.
- A^{-1} will refer to the inverse of A, assuming that it exists. $A^{-1}A = AA^{-1} = I$
- An orthonormal matrix U is one whose transpose is equal to its inverse. $U^TU = UU^T = I \iff U$ orthonormal.
- $\langle a, b \rangle$ will denote the inner product of vectors $a, b \in \mathbb{R}^m$. $\langle a, b \rangle = \sum_{i=0}^{m} a_i b_i = a^T b$
- $\|a\|$ will denote the Euclidean norm of a vector $a \in \mathbb{R}^m$. $\|a\| = \sqrt{\sum_{i=0}^{m} a_i^2}$
- $\|A\|$ will denote the matrix norm induced by the Euclidean norm: $\|A\| = \sup_{\|x\|=1} \|Ax\|$
- e_i will denote the vector with $[e_i]_j = 1$ if $i = j, [e_i]_j = 0$ otherwise.

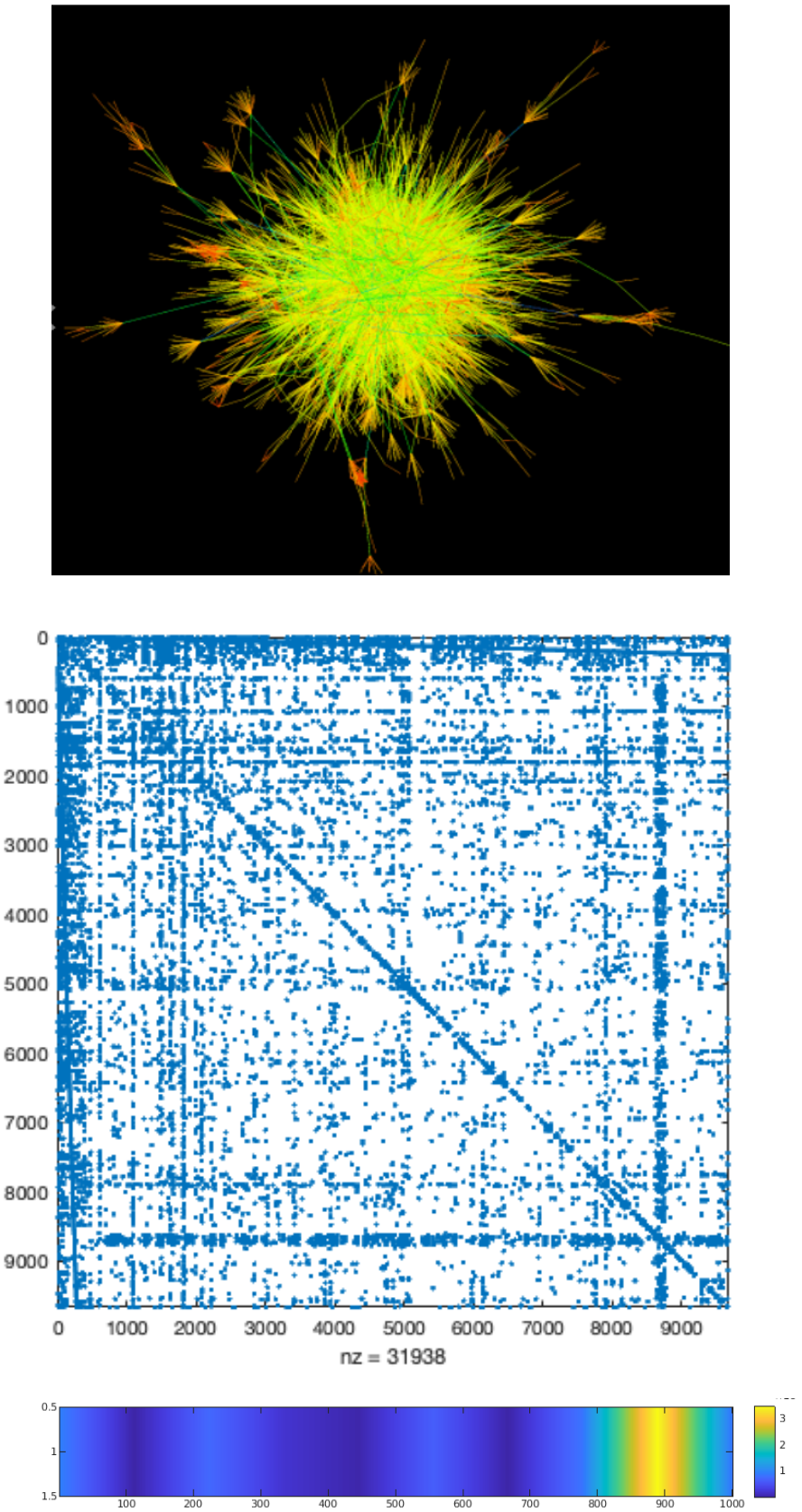


Figure 1: A graph (top), its matrix representation (middle), and an exponential measure of the centrality of its nodes (bottom). Note the strong block line in the square matrix corresponds to a high score on the colorchart. (California.mtx)

1 Preliminaries

1.1 Matrices and Matrix Functions

This project will only consider real, square, symmetric matrices $A \in \mathbb{R}^{n\times n}, A^T = A$.

1.1.1 Eigendecomposition of Real Symmetric Matrices

Recall that the eigendecomposition of a matrix A finds the *eigenvectors*, those vectors v_i for whom multiplication by A is equivalent to multiplication by a scalar:

$$Av_i = \lambda_i v_i \quad \text{or} \quad AV = V\Lambda \tag{2}$$

where the λ_i are called *eigenvalues*. Each λ_i with its corresponding v_i we shall refer to as an *eigenpair*: (λ_i, v_i). For convenience we will assume that all the eigenvectors are normalized with respect to the Euclidean norm; that is $\|v_i\| = \sqrt{\sum_{k=1}^{n} v_{ki}^2} = 1$. Where $V \in \mathbb{R}^{n\times n}$ is the matrix of eigenvectors and $\Lambda \in \mathbb{R}^{n\times n}$ is the diagonal matrix of eigenvalues. This gives the decomposition $A = V\Lambda V^{-1}$, with the nice property $A^k = V\Lambda^k V^{-1}$.

Theorem 1.1. *All real symmetric matrices are diagonalizable, $A = V\Lambda V^T$, where V, Λ are the eigenvectors and eigenvalues of A, respectively.*

See [6] for a proof.

Theorem 1.2. *Symmetric matrices have orthogonal eigenvectors.*

Proof: For symmetric A take two eigenvectors v_i, v_j with associated eigenvalues λ_i, λ_j.

(i): $\lambda_i \neq \lambda_j$

$$\begin{aligned}
&\lambda_i\langle v_i, v_j\rangle = \langle \lambda_i v_i, v_j\rangle \\
&= \langle Av_i, v_j\rangle \quad \text{since } v_i \text{ is an eigenvalue} \\
&= \langle v_i, Av_j\rangle \quad \text{since } \langle Ax, y\rangle = \langle x, A^T y\rangle, \text{ and } A = A^T \\
&= \langle v_i, \lambda_j v_j\rangle = \lambda_j\langle v_i, v_j\rangle \\
&\implies (\lambda_j - \lambda_i)\langle v_i, v_j\rangle = 0 \\
&\implies \langle v_i, v_j\rangle = 0
\end{aligned}$$

Therefore, eigenvectors have distinct eigenvalues $\implies \langle v_i, v_j\rangle = 0 \iff v_i, v_j$ are orthogonal.

(ii): $\lambda_i = \lambda_j$.

Assume that matrix A has $k \leq n$ distinct eigenvalues. By (i) we know that the associated eigenvectors span k mutually orthogonal eigenspaces. We can construct an orthogonal basis B_i for each of these eigenspaces, where $\text{rank}(B_i) \leq \#$ (eigenvectors associated with λ_i). The union of these bases $B = \cup^k B_i$ will be a subspace, and since symmetric matrices are diagonalizable, B must have full rank, making B an orthogonal basis for $\mathbb{R}^n$. ■

Therefore we can find an eigendecomposition $A = V\Lambda V^{-1}$ such that V is orthonormal (scaling the v_i to unit length), meaning $V^{-1} = V^T$

So our eigendecomposition becomes:

$$A = V\Lambda V^T \tag{3}$$

1.1.2 Upper Hessenberg and Tridiagonal Matrices

A Hessenberg matrix H, is one whose entries are all zero below the subdiagonal. That is: $i > j+1 \implies h_{ij} = 0$. This project will compute a partial factorization of $A = UHU^T$ where U is orthogonal. When $A = A^T$ this Hessenberg decomposition becomes:

$$\begin{aligned} & A = A^T \\ & \iff UHU^T = (UHU^T)^T \\ & \iff UHU^T = UH^TU^T \\ & \iff H = H^T \end{aligned}$$

Meaning that H is tridiagonal. In other words for $i < j-1$ or $i > j+1 \implies h_{ij} = 0$. To unambiguously refer to the tridiagonal H we use the letter T.

1.1.3 Projection

A projection is a transformation of a vector v onto a subspace with range(P): $v \to Pv$. Typically $v \notin \text{range}(P)$, however if $v \in \text{range}(P)$, it will be unaltered by the projection P. For this reason $P(Pv) = Pv$, meaning $P^2 = P$. Typically P has rank less than n, making projection a *rank reducing operation*. A typical example projects a vector onto a single line, q. Here P is constructed as $P = qq^T$. In order for $P = P^2$ we must have $qq^Tqq^T = qq^T$, meaning that the inner dot product qq^T must equal one, imposing the condition that $\|q\| = 1$. An orthogonal projection is one where Pv is orthogonal to the path along which v is projected to arrive at Pv: $Pv \perp Pv - v$. See figure 2.

Take some orthonormal basis Q for a subspace $V \subseteq \mathbb{R}^n$, where $\dim(V) \leq n$, a projection P onto V can be constructed by multiplying the basis by its transpose: $P = QQ^T$. Since if $y \in V, y = Px$ for some x, $Py = P(Px) = QQ^TQQ^Tx = QQ^Tx = Px = y$, assuming that Q is orthonormal ($Q^TQ = I$).

When projecting some x onto range(P) we typically want to minimize the distance $\|Px - x\|$, so that Px can be used a good approximation to x. This distance is minimized by orthogonal projection so we will henceforth use orthogonal projectors only.

1.1.4 Matrix Functions

A function f applied to a matrix A is defined as

$$f(A) = Vf(\Lambda)V^{-1} \tag{4}$$

Where:

$$f(\Lambda) = \begin{bmatrix} f(\lambda_1) & 0 & \dots & 0 \\ 0 & f(\lambda_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & f(\lambda_n) \end{bmatrix}$$

If A is *diagonalizable*. We will only consider the case where A is diagonalizable, since all symmetric matrices are diagonalizable. See below:

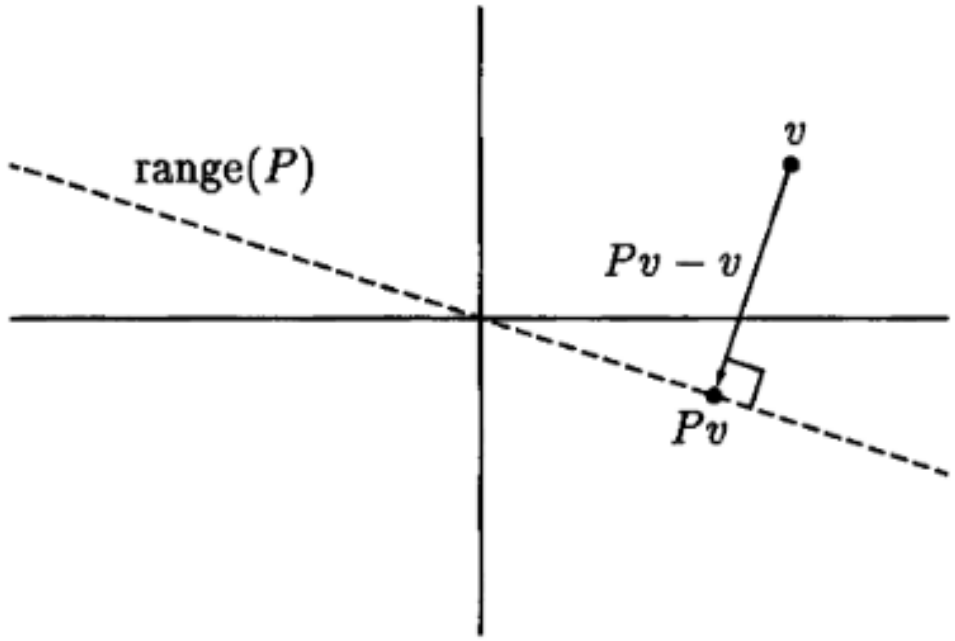


Figure 2: An orthogonal projection of v onto range(P): $v \to Pv$. Source [4].

1.2 Graphs and Graph Representation

A graph $G = (V, E)$ is a set of n nodes and $|E|$ edges, where $E = \{(i, j) \,|i, j \in V\}$. If $(i, j) \in E$ then we say that nodes i, j are connected by edge (i, j), meaning that something can walk or be transmitted from node i to node j in a single step. If $(i, j) \in E \iff (j, i) \in E$ we say that the graph G is undirected, meaning that all edges are *two way streets*. This project will only consider *undirected* graphs. Any edge (i, j) can have an associated weight $w_{1<k<|E|}$, which may represent the cost of travelling from nodes (i, j) or the strength of the connection between nodes i, j. This project will only consider unweighted graphs, or weighted graphs where $w_k = 1, \forall k$. In unweighted graphs all edges are considered equal. The *degree* of a node i is the number of nodes that node i. A *walk* of length k is a list of nodes $n_1, n_2, \ldots, n_k$ where subsequent nodes are connected, $(n_{j-1}, n_j) \in E, \forall j$.

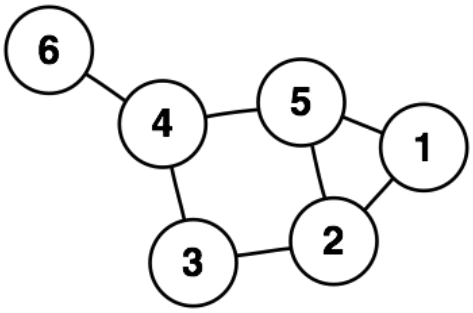
A graph G can be represented as an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases} \tag{5}$$

For a weighted graph $a_{ij} = w_{ij}$, the weight of edge (i, j). Since this project deals with undirected, unweighted graphs, the adjacency matrices will be symmetric ($A^T = A$), with binary entries $a_{ij} \in \{1, 0\} \forall i, j$.

1.3 Graph Analysis and Centrality

One of the key aims of graph analysis is to measure the *centrality* of nodes in a graph. Centrality can be defined and measured in myriad ways.



(a) A simple graph G

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(b) G's adjacency matrix

1.3.1 Centrality Measures

Degree centrality simply uses the degree of a vertex as a measure of how central it is. This method is simplistic but very easy to compute: $[A\mathbf{1}]_i$ tells us the degree of node i.

Eigenvector centrality uses the magnitude of the i^{th} entry of the leading eigenvector v_1 as a proxy by which to measure the centrality of node i. Intuitively, a large $[v_1]_i$ value is caused by a high degree for node i as well as a high degree for the nodes that it shares edges with.

Closeness Centrality gives each node a score inversely proportional to its average minimum distance to all other nodes.

The *Betweenness Centrality* [1] of a given node is defined as

$$C_B(i) = \sum_{j \neq i} \sum_{k} \delta_{jk}(i) \tag{6}$$

Where

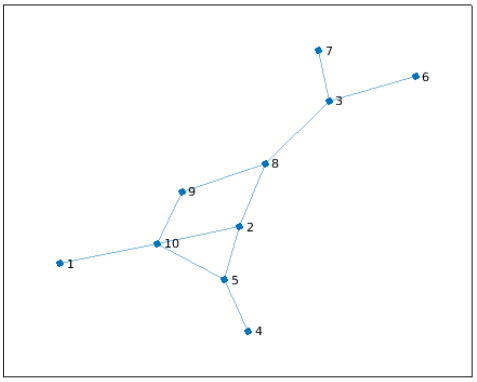
$$\delta_{jk}(i) = \frac{\text{\# shortest paths between j, k containing i}}{\text{\# shortest paths between j, k}}$$

Intuitively, $\delta_{jk}(i)$ tells us the probability of a signal from j to k passing through node i. Therefore the betweenness centrality (once the vector C_B is normalized) will give the probability of any message passing through a given node (assuming that messages between nodes i, j follow a shortest path between nodes i, j [there may be many shortest paths]).

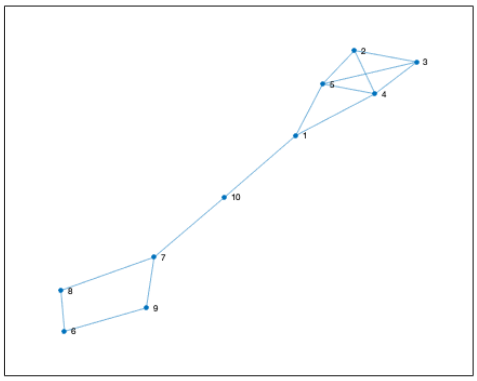
Other measures of centrality include *stress centrality*, Google's famous *Pagerank* centrality, *exponential centrality* (or *total node communicability*, which we will examine in section 1.4) as well as many others. See [3].

Simple measures of centrality, such as *degree centrality* will only be a good metric if the graph in question is relatively balanced with a single big cluster, like in Figure 4a. Take instead a graph such as figure 4b, where we have two clear clusters, and a bridge connecting them. It is clear that the cluster of nodes 1-5 are very well connected, however node 10 also seems to serve a vital purpose in the overall connectivity of the graph. Each centrality measure will give a different appraisal of the graph, and in using many such measures we can get a good picture of the entire graph.

It is clear from Table 1 that most measures agree on the most central nodes of the graph in Figure 4a. Closeness and betweenness centrality clearly have a different measure of centrality



(a) A relatively balanced graph



(b) A dumbbell-shaped graph, with multiple clusters

Node	Degree	Pagerank	Closeness	Betweenness	Eigenvector	Exponential
1	1	0.0504	0.0370	0	0.0690	7.7820
2	3	0.1235	0.0588	12.0000	0.1730	18.9721
3	3	0.1487	0.0476	15.0000	0.0655	11.3401
4	1	0.0516	0.0357	0	0.0562	6.6374
5	3	0.1292	0.0500	8.0000	0.1531	16.6738
6	1	0.0571	0.0345	0	0.0241	5.3637
7	1	0.0571	0.0345	0	0.0241	5.3637
8	3	0.1290	0.0588	18.5000	0.1304	16.1898
9	2	0.0869	0.0526	4.0000	0.1168	13.3347
10	4	0.1663	0.0526	10.5000	0.1879	20.4108

Table 1: Comparing measures of centrality for the graph in Figure 4a. The most centrally-ranked node is highlighted for each measure. Computed with MATLAB's `centrality()` function.

than the other measures. They seem to locate nodes in the *middle* of a graph, which may or may not be well connected to the immediate environment. Now to examine how these metrics shall measure the dumbbell-shaped graph.

From Table 2 we see the startling variance in each centrality measure's output. Nodes 1, 4, 5, 7, and 10 are each ranked as the most central by at least one of the measures, which is half of the graph! It is clear that the denser subgraph of nodes 1-5 make most centrality measures biased in their favour. Although only betweenness and closeness centrality highlight the importance of node 10, it is encouraging that most measures identify node 10 as more central than node 7, which has a higher degree.

This does not suggest that betweenness and closeness centrality are more or less *correct* than any others, but that their measure of centrality may more easily identify bottlenecks such as node 10 in this example. This occurs since the former is concerned with shortest paths passing through a given node, and the latter is concerned with distances to all other nodes. It is clear that node 10 scores highly in both of these metrics. All measures of centrality give different

Node	Degree	Pagerank	Closeness	Betweenness	Eigenvector	Exponential
1	3	0.1003	0.0526	20	0.1395	24.5539
2	3	0.0936	0.0357	0	0.1788	28.1522
3	3	0.0936	0.0357	0	0.1788	28.1522
4	4	0.1225	0.0455	6	0.2108	33.8338
5	4	0.1225	0.0455	6	0.2108	33.8338
6	2	0.0899	0.0303	0.5	0.0038	8.1754
7	3	0.1232	0.0476	18.5	0.0178	12.5824
8	2	0.0881	0.0370	3.5	0.0064	9.0379
9	2	0.0881	0.0370	3.5	0.0064	9.0379
10	2	0.0783	0.0526	20	0.0468	12.9328

Table 2: Comparing measures of centrality for the graph in Figure 4b. Computed with MATLAB's `centrality()` function.

insights in the analysis of graphs. With a combination of multiple centrality measures it is possible to get a nuanced and multi-faceted appreciation of the structure of a graph.

1.3.2 Performance of Spectral and Traversing Graph Centrality Measures

Spectral Graph Methods are those that choose to analyze a graph through its linear-algebraic properties, such as eigenpairs, as well as the characteristic polynomial. Although matrix functions rarely explicitly compute eigenvalues or eigenvectors, I will also consider them a spectral method. Eigenvector, Pagerank (which is perhaps both spectral *and* traversing! [9]), and exponential centrality measures I will also refer to as spectral methods.

Betweenness and closeness centrality rely instead on graph traversal techniques: walking from one node to the next and so on. I will consider them *Graph Traversal Methods*

While it is valuable to use many centrality measures in order to fully analyse a graph, quite often this is an unaffordable luxury due to the time and space requirements of each algorithm.

Calculating degree centrality is the computationally cheapest procedure, since computing $A\mathbf{1}$ runs in $O(|E|)$ time, assuming that A is sparse. Google's Pagerank runs in $O(|E| * k)$ time [9], where k is a chosen number of iterations. Eigenvector centrality can also be computed in $O(|E| * k)$ time, by multiplying the sparse A by some vector v, for some chosen k iterations. Computing exponential centrality (*total node communicability*) can be done in $O(k * (n + |E|))$ steps, as will be shown in **??**. [3] proposes an algorithm for computing betweenness centrality that runs in $O(n * |E|)$ time. Closeness centrality runs in $O(n * (|E| + 1)$ time although there exist faster approximations [7].

However, time complexity is not the only factor that can influence runtime for graph algorithms. [8] outlines several problems that hinder optimisation efforts, as well as the difficulty in parallelizing graph algorithms. Among the difficulties are:

- **Unstructured Problems:** Graphs lack predictable structures that can be cleanly partitioned among processes, this can result in complex halo-exchange routines or occasionally

rule out parallelization altogether.

- **Poor Locality:** Graph algorithms that call for graph traversals do not exploit data locality, since edges or nodes cannot be reliably arranged such that neighbouring nodes are physically close in memory. This results in poor cache performance.
- **Low arithmetic intensity:** This is especially true for algorithms using traversal techniques, where many loads may be performed to walk around the graph, with fewer arithmetic operations.

Spectral Methods by comparison, use Linear Algebraic methods that are far more regularly structured than most graph algorithms, which allows them to give better performance as well as easier parallelization than traversal methods.

1.4 The Matrix Exponential as a Centrality Measure

The matrix exponential is defined as:

$$e^A = I + A + A^2/2! + A^3/3! + \ldots = \sum_{k=0}^{\infty} A^k/k! \tag{7}$$

Or alternately:

$$e^{tA} = I + tA + t^2A^2/2! + t^3A^3/3! + \ldots = \sum_{k=0}^{\infty} t^kA^k/k! \tag{8}$$

For some $t \in \mathbb{R}$.

The matrix exponential arises frequently in the study of many systems as the solution of the ordinary differential equation:

$$x'(t) = Ax(t) \tag{9}$$

Here it is clear that the solution vector is:

$$x(t) = e^{tA}x_0 \tag{10}$$

The ubiquity of equation 9 makes studying the solution vector $e^{tA}x_0$ vitally important; however, this project will consider the matrix exponential through the lens of graph analysis, which offers an additional interpretation.

To understand the significance of the matrix exponential for adjacency matrices, we must recognize that in the matrix A^k, the a_{ij} entry tells us the number of paths of length k from node i to node j. To see this first consider the case for A^1. An entry a_{ij} of A tells us whether there is an edge between nodes i, j—in other words, the number of walks of length one between nodes i, j. When we multiply A by itself, the entries are:

$$[A^2]_{ij} = \sum_{k=1}^{n} a_{ik}a_{kj} \tag{11}$$

where $a_{ik}a_{kj} = 1 \iff a_{ik} = 1$ and $a_{kj} = 1$. In other words:

$$a_{ik}a_{kj} = 1 \iff \exists \text{ a walk of length 2 between nodes } i, j \text{ through node } k \tag{12}$$

Therefore the sum of all such terms (11) will tell us the number of walks from node i to node j that go through some other node k. By induction we can see that $[A^m]_{ij}$ holds the number of walks of length m from node i to node j.

In this light, since the matrix exponential (equation 7) is a weighted sum of *all* powers of A, a given entry $[e^A]_{ij}$ represents the weighted sum *all* possible walks from node i to node j. Here the weights (the $1/k!$ coefficients) allow shorter walks to contribute more to e^A than the longer ones. A high relative value of $[e^A]_{ij}$ signifies to a high degree of communicability between nodes i, j, whereas a low relative value signifies a low degree of communicability. This also means that $\sum_{k=0}^{N} A^k/k! \approx e^A$ for N relatively large, although we would not calculate e^A explicitly like this.

In the matrix e^A, the diagonal $[e^A]_{ii}$ tells us the weighted sum of all closed loops from node i to itself. The non-diagonal components $[e^A]_{ij}$ give a measure of *communicability* between nodes i, j, or how many paths there are between the two nodes, determining how easily information flows between nodes i, j.

If we apply the exponential of our matrix onto a vector of ones, each entry $[e^A\mathbf{1}]_i = \sum_{k=1}^{n}[e^A]_{ik}$ gives a measure of the number of paths from node i to all other nodes. This measure may be referred to as the *Total Node Communicability* [1].

$$TC(i) = [e^{\beta A}\mathbf{1}]_i \tag{13}$$

The *Total Node Communicability* gives us a good understanding of how easy it is for a given node to transmit information in a graph. Identification of nodes that communicate poorly is of utmost importance in the analysis and control of all manner of networks. This project will compute the Total Node Communicability of large, sparse adjacency matrices.

1.5 Krylov Methods

Matrix computations come at a high cost for large matrices. If we want to compute $f(A)x$, a thorough method would make computations of the same dimension as A. For many algorithms this is not only computationally infeasible, but also *unnecessary*.

Krylov Methods, instead of dealing with large rank n systems, *project* a given problem onto a subspace of dimension r, where r is chosen in advance. The *Krylov Subspace* $\mathcal{K}_r(A, v)$ is the reduced-rank subspace and is built by a matrix A, vector v and a chosen dimension r. Call Q_r a basis for $\mathcal{K}_r$. Intuitively, this is possible because the answer that we desire is a vector, which has rank of one; it ought to be feasible to make a good approximation to a rank one vector by using significantly less than n vectors.

$$\mathbb{R}^n \xrightarrow{\text{projection}} \mathcal{K}_r$$
$$f(A)x \xrightarrow{\text{projection}} Q_r y.$$

Where y is some vector of coefficients for the basis vectors in Q_r, which is chosen so that $\|f(A)x - Q_r y\|$ is small.

The *Krylov Subspace* is defined as:

$$\mathcal{K}_r(A, v) = \text{span}\left\{b, Ab, A^2, \ldots, A^{r-1}b\right\} \tag{14}$$

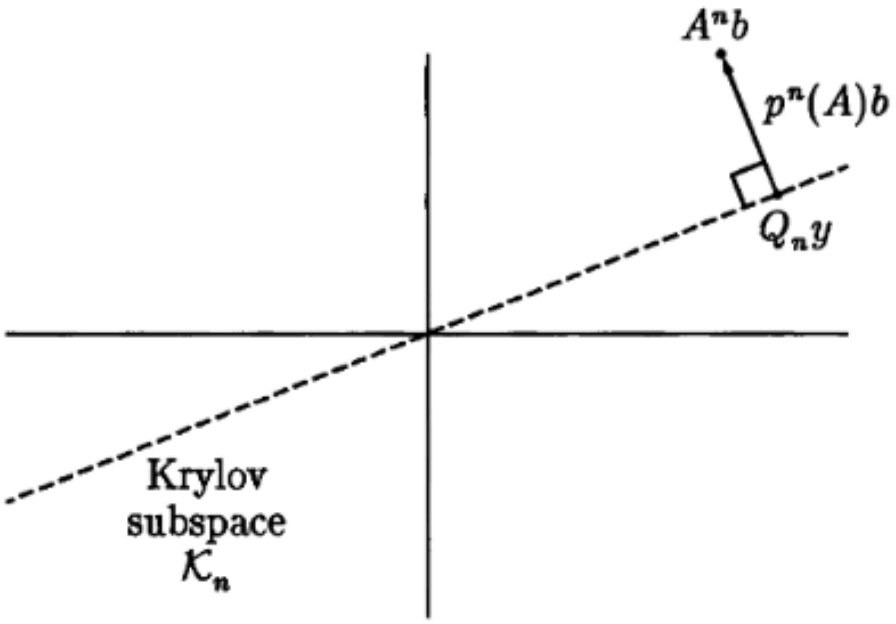


Figure 5: Here $Q_n y$ is the linear combination of the Krylov subspace vectors, where y is chosen to minimize $\|f(A)x - Q_n y\|$. $p^n(A)b$ is the polynomial of degree n that extends the range of $\mathcal{K}_n$. (Note that this picture uses n as the dimension of the Krylov subspace, whereas this project will use r to refer to the dimension of the Krylov subspace.) Source [4].

This subspace has rank $r \ll n$, chosen to be much less than n. Note that any $u \in \mathcal{K}_r(A, v)$ is a vector that can be written as a some polynomial of A, with v given:

$$u = p(A)v = y_o v + y_1 Av + y_2 A^2 v + \ldots + y_{r-1} A^{r-1} v$$

When we are approximating $f(A)x$, we always build the Krylov subspace with x as the starting vector. That is we always seek to find an approximation $Q_r y$ to $f(A)x$, where Q_r is the basis for $\mathcal{K}_r(A, x)$. While there are many $f(A)x \in \mathbb{R}^n, f(A)x \notin \mathcal{K}_r(A, x)$, we can generally find a very high quality approximation $Q_r y$ to $f(A)x$ that *is* in the Krylov subspace: $\|f(A)x - Q_r y\|$ is small. The most-studied function f for which $f(A)x$ is typically computed is $f(A) = A^{-1}$. There are a multitude of Krylov-based algorithms which seek to find $q(A)x \approx A^{-1}b$ with the utmost speed and stability, see [10].

As r increases, the vectors converge $A^{r-1}b \to v_1$, the leading eigenvector of A. While this makes eigenvector centrality computations straightforward (see section 1.3.1), it does not make for a well-behaved subspace. This is because the orthogonal component of each additional vector can easily be rounded out or corrupted due to the high degree of dependence among the $A^m b$ vectors, for large m.

1.5.1 Arnoldi's Method and the Lanczos Algorithm

Instead of using a naïve basis for $\mathcal{K}_r$ as in equation 14, it is beneficial to build an orthogonal basis for $\mathcal{K}_r$ by orthogonalizing each new vector against all the previous basis vectors. Arnoldi's method does exactly this.

Choose v_1 such that $\|v_1\| = 1$ **for** $j = 1, 2, \ldots, r$ **do**
  $h_{ij} = \langle av_j, v_i \rangle$ for $i = 1, 2, \ldots, j$
  $w_j = av_j - \sum_{i=1}^{j} h_{ij} w_j$
  $h_{j+1,j} = \|w_j\|$
  if $h_{j+1,j} = 0$ then stop
  $v_{j+1} = w_j / h_{j+1,j}$
end

Algorithm 1: Arnoldi's method

Arnoldi's method constructs an orthogonal basis $Q_r \in \mathbb{R}^{n \times r}$ of the Krylov subspace, as well as the upper-Hessenberg matrix H (see properties in 1.1.2). We can write this algorithm in matrix form as:

$$AQ_r = Q_r H_r + w_r e_r^T$$

Where each multiplication of A adds another dimension to the subspace. If we omit the added column vector, we get the projection of A onto range(Q_r):

$$H_r = Q_r^T A Q_r$$

Since in this project we are concerned with symmetric $A = A^T$, this equation becomes:

$$\begin{aligned} H_r^T &= (Q_r^T A Q_r)^T \\ H_r^T &= Q_r^T A^T Q_r \\ H_r^T &= Q_r^T A Q_r \text{ since } A = A^T \\ H_r^T &= H_r \end{aligned}$$

Meaning that the upper-Hessenberg is actually tridiagonal: $h_{ij} = 0$ if $i > j + 1$ or $i < j - 1$. Therefore it is only necessary to compute $h_{j-1,j}, h_{jj}, h_{j,j+1}$, and in fact $h_{j-1,j} = h_{j,j+1}$, by the symmetry of H_r. This reduces the work required to compute H_r drastically, where before we needed to compute $\sum^r (i+1) = r(r+1)/2 + r$ values, now we only need to compute $2r$ values. This simplified algorithm is known as the *Lanczos Method*.

```
function [alpha, beta,Q] = Lanczos(A, v, k)

Q(:, 1) = v/norm(v);
for j=1:k
    v = A*Q(:,j);
    alpha(j) = Q(:,j)'*v;
    v = v-alpha(j)*Q(:,j); % Orthogonalizing v against the previous basis vector
    if j > 1
        v = v-beta(j-1)*Q(:,j-1); % Orthogonalizing v against the second
   previous basis vector
```

```
    end
    if j < k
        beta(j) = norm(v);
        Q(:,j+1) = v/beta(j);
    end
end
```

Listing 1: MATLAB code for the Lanczos iteration

Here the diagonal is represented by the vector $\alpha \in \mathbb{R}^r$, and the sub- and super-diagonal is represented by the vector $\beta \in \mathbb{R}^{r-1}$. At the end of the algorithm we have the projection $A \to QTQ^T$, where $T \in \mathbb{R}^{r \times r}$ is the tridiagonal form of H and $Q \in \mathbb{R}^{n \times r}$ is an orthogonal basis for $\mathcal{K}_r$.

For a fuller treatment of the Arnoldi algorithm see [4].

2 Approaches to Compute $\exp(A)x$

There are many ways of computing the matrix exponential. A direct method would follow the formula outlined in section 1.1.4: $e^A x = V e^\Lambda V^T x$. This is convenient for symmetric A, since we are guaranteed to have orthogonal eigenvectors, however this method can break down for non-symmetric matrices, when there is a chance of linear dependencies among eigenvectors. However, for symmetric A this still necessitates a full eigendecomposition, which is an $O(n^3)$ operation, and also requires storing the V in memory, which requires $O(n^2)$ space. V is liable to be several orders of magnitude bigger than A, since V will be dense, whereas A is assumed to be sparse, making this operation only suitable for small n.

A Taylor series expansion, as seen in equation 7, is unsuitable since repeated multiplication of A is liable to introduce errors if A is poorly conditioned, that is if $\|A\|\|A^{-1}\|$ is large. Moreover the convergence of $I + A + A^2/2! + \ldots \to$ is often slow, meaning that convergence is often outpaced by numerical errors.

[5] mentions that a generally robust way of computing e^A is by using a *squaring and scaling* method, which computes e^A by first scaling A so that $\|A\| \approx 1$, computing $e^{A/m}$ with a Padé Approximation (we will omit the details), and then repeatedly squaring $(e^{A/m})^m$ in order to get e^A. This has far better stability than the Taylor expansion since fewer multiplications are involved, and since the condition of A is replaced by the better conditioned A/m. This method still requires $O(n^3)$ operations and $O(n^2)$ space requirements although it does result in a full matrix to analyze: $e^A \in \mathbb{R}^{n \times n}$.

When A gets large, matrix-matrix computation and non-sparse matrix storage become increasingly difficult. It becomes necessary to reduce the scope of a computation, instead of seeking to find $e^A \in \mathbb{R}^{n \times n}$ it is better to make do with $e^A x \in \mathbb{R}^n$, which in itself can give a valuable centrality measure for a given A, the *Total Node Communicability*.

We can use Krylov Methods (section 1.5 to approximate this vector $e^A x$ with a high degree of accuracy. The approach proceeds as follows for symmetric A, using the normalized starting

vector as $q_1 = x/\|x\|$.

$$\begin{aligned} A &\approx QTQ^T && \text{Projection of } A \text{ onto } \mathcal{K}_r, \text{ using the Lanczos decomposition.} \\ A &\approx Q(V\Lambda V^T)Q^T && \text{(Orthogonal) Eigendecomposition of the tridiagonal } T = V\Lambda V^T. \\ f(A) &\approx f(Q(V\Lambda V^T)Q^T) && \text{Apply } f \end{aligned}$$

$(QV)\Lambda(QV)^T$ is a sort of truncated Eigendecomposition (truncated in the sense that QV, the eigenvector matrix, is non-square) of the projection of A onto $\mathcal{K}_r$. From section 1.1.4, the matrix function is defined as $f(B) = V_B f(\Lambda_B) V_B^T$ for symmetric B. So we can apply f to A as:

$$f(A) \approx QVf(\Lambda)V^TQ^T \tag{15}$$

Now we must apply this approximation to x.

$$f(A)x \approx QVf(\Lambda)V^TQ^Tx$$

Note that since we have built Q starting with $q_1 = x/\|x\|$, and since Q is orthonormal ($Q^Tq_1 = e_1$), $Q^Tx = Q^Tq_1\|x\| = \|x\|e_1$.

So the equation becomes:

$$f(A)x \approx QVf(\Lambda)[V^T]_1 \tag{16}$$

Where $[V^T]_1 = V^Te_1$ is the first column of V^T, or the first entry of each eigenvector. This is the final form of the approximation that we will use to calculate e^Ax, or any other $f(A)x$.

Note that equation 15 alone is not a suitable approximation to $f(A)$. It is only a reduced rank construct in order to calculate $f(A)x$, and not to represent $f(A)$ accurately. This is because $\mathcal{K}_r$ has been constructed explicitly to map the relationship between A and x, and not to represent the relationship that A may have with *any* given vector v.

3 Implementation

3.1 Sparseness and Modified CSR

In many real world networks, the average degree per node is far less than the number of nodes in the network. Therefore, naively storing A as an adjacency matrix as in section 1.2 with an entry for every single possible edge is very memory inefficient, since a very large proportion of the stored data will be zeros. With this in mind, sparse matrix storage schemes aim to reduce the memory footprint of a matrix from $O(n^2)$ to $O(|E|)$, by only storing the nonzero values in memory.

This project uses a modified CSR (Compressed Sparse Row) storage scheme (see [11] for a full description). CSR format stores A in three arrays:

- AA: Holds the values at the nonzero entries of the matrix. A 1d array of length nnz, the number of nonzero entries in A.

- IA: Indicates the index at which the j^{th} row begins in AA and JA. Has length $n+1$.
- JA: Indicates the column indexes at which a nonzero element occurs. Length nnz.

In figure 6, for instance, row 1 (0-indexed) starts at $idx = IA[1] = 2$, and ends at $idx = IA[2] = 5$, meaning it contains 3 values:

- $AA[2] = 7$ at entry $(1, JA[2]) = (1, 0)$.
- $AA[3] = 1$ at entry $(1, JA[3]) = (1, 2)$.
- $AA[4] = 5$ at entry $(1, JA[4]) = (1, 4)$.

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 2 \\ 7 & 0 & 1 & 0 & 5 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(a) A in full format

$$AA = [3, 2, 7, 1, 5, 1, 1]$$
$$IA = [0, 2, 5, 7]$$
$$JA = [1, 4, 0, 2, 4, 1, 3]$$

(b) A in CSR format

Figure 6: Full and CSR storage schemes

However, this project will only compute the exponential for unweighted graphs, meaning that AA contains only ones. For this reason we do not need to hold AA in memory—only the arrays IA, JA are necessary. It is a straightforward transition to adapt the code to hold weighted graphs, but this limits the size of graph we can hold in memory, since memory requirements for A go from $n + |E|$ to $n + 2|E|$.

3.2 Computing $e^A x$

Computing $e^A x$ for a large, sparse A is divided into a number of tasks, in order to generate the expression shown in 16.

- Compute a Lanczos decomposition. $A \approx QTQ^T$. Using the algorithm shown in Listing 1. This is achieved by the class `lanczosDecomp`. See `serial/lib/lanczos.cc` and `serial/lib/lanczos.h`.
- Compute the Eigendecomposition of T: $A \approx QV\Lambda V^T Q^T$. This is done with the LAPACKE routine `LAPACKE_dstevd`. See `serial/lib/eigen.cc` and `serial/lib/eigen.h`.
- Apply f to Λ and multiply out $y = QVf(\Lambda)[V^T]_1\|x\|$ to obtain $y \approx f(A)x$. See `serial/lib/multiplyOut` and `serial/lib/multiplyOut.h`.

4 Numerical Properties

4.1 Sensitivity of the Problem

Define the condition of the transformation $x \to Ax$ bounded by: $\text{cond}(A) \leq \|A\|\|A^{-1}\|$. Since we use the condition of A to describe the conditioning of the matrix vector multiplication Ax, we shall use $\text{cond}(e^A)$ to describe the conditioning of our problem $e^A x$.

We are interested in finding a bound for the expression:

$$\phi(t) = \frac{\|e^{t(A+E)} - e^{ta}\|}{\|e^{tA}\|}$$

Which represents the sensitivity of the problem with respect to some perturbation E.

$$\phi(t) \leq \|E\|e^{\|E\|t}$$

Which we can use since A is always normal when A is symmetric. Van Loan mentions that these bounds are as small as can be expected, making $e^A x$ a well conditioned problem for A symmetric.

4.2 Lanczos Approximation Error

[14] outlines that the error of the Lanczos approximation y to $f(A)x$ can be bounded by:

$$\|f(A)x - y\| \leq 2 \cdot \|x\| \cdot \min_{\text{polynomial } p \text{ with degree } <k} \left(\max_{u \in [\lambda_{\min}(A), \lambda_{\max}(A)]} |f(u) - p(u)| \right) \tag{17}$$

Note that this bound for the Lanczos error does not include the term x (treating $\|x\|$ as a scalar). So this bound for $\|f(A)x - y\|$ is really a bound for $\|f(A)v - y\|$, for any given v assuming that $\|v\| = \|x\|$. However, we have explicitly constructed $\mathcal{K}_r$ to map the relationship between A and a specific x, meaning this bound is likely to far exceed observed errors.

Analytically, we can consider $x = \sum \alpha_i v_i$ where the v_is are the eigenvectors of A which form a basis for $\mathbb{R}^n$, and the α_i are some coefficients of these basis vectors. We know that the α_i are unique for a given basis of eigenvectors. Then perform a full eigendecomposition of A (we know that one exists from section 1.1.1) and express the Lanczos approximation to $f(A)x$ as $p(A)x$, a polynomial of A acting on x.

$$\begin{aligned} \|f(A)x - p(A)x\| &= \|Vf(\Lambda)V^T x - Vp(\Lambda)V^T x\| \\ &= \|V(f(\Lambda) - p(\Lambda))V^T x\| \end{aligned}$$

Note that $V^T x = V^T \sum \alpha_i v_i$ and $V^T v_j = e_i$ since V is orthonormal (by A symmetric). So $V^T \sum \alpha_i v_i = e_1\alpha_1 + \ldots + e_n\alpha_n = \vec{\alpha}$. Making our error term:

$$\|f(A)x - p(A)x\| = \|V(f(\Lambda) - p(\Lambda))\vec{\alpha}\|$$

Since V is orthonormal we can discard the V term, making this is equal to:

$$\|f(A)x - p(A)x\| = \|(f(\Lambda) - p(\Lambda))\vec{\alpha}\| = \begin{bmatrix} (f(\lambda_1) - p(\lambda_1))\alpha_1 \\ \vdots \\ (f(\lambda_n) - p(\lambda_n))\alpha_n \end{bmatrix}$$

Meaning that in order for $\|f(A)x - p(A)x\|$ to be large, x needs to point considerably (corresponding to a large α_i term) in the direction of an eigenvector that is not well *mapped* by the Krylov approximation, $p(A)x$. However this is unlikely, since $p(A)x$ is built by multiplying A and x; surely the generated Krylov subspace will have the v_i direction mapped relatively well, if $\langle x, v_i \rangle$ is large.

The accuracy of the Lanczos approximation will be explored in the next section. This high error bound (Equation 17) ought to be kept in mind, but also the intuition that the approximation ought to behave far better than this bound may suggest.

4.3 Comparing With an Analytic Answer

In order to check the numerical accuracy of the Lanczos approximation, it was necessary to check the obtained answer vector with a precomputed analytic answer.

See `serial/tests/numerical_test.cc` for implementation. Using this method it was also possible to check the convergence of the Lanczos approximation for different values of r. Note that an analytic answer will, like the Lanczos approximation, be computed numerically and so may be considered a *pseudo-analytic* answer, because of the presence of rounding errors. However these rounding errors are on the order of ϵ_{mach} which is tolerable for our purposes.

In order to find an analytic answer to $e^A x$, we could use the definition of matrix functions outlined in section 1.1.4, that is: $f(A) = Vf(\Lambda)V^{-1}$. However, this requires a full eigendecomposition, which limits the size of adjacency matrix A that we can deal with, since an eigendecomposition requires $0(n^3)$ operations and we will need to store the dense V which requires $O(n^2)$ memory.

A more elegant approach is to take the first k eigenpairs. We can use MATLAB to find the first k eigenpairs using the `eigs()` function. Once we have said eigenpairs, we can construct our starting vector as a weighted combination of the eigenvectors:

$$x = \sum_{i=0}^{k} \alpha_i v_i \tag{18}$$

Where the $\alpha_i \in \mathbb{R}$ values are randomly generated weights $0 < \alpha_i < 1$, and v_i is the i^{th} eigenvector. Once we have this starting vector, we can use the formula $f(A) = Vf(\Lambda)V^T$ along with some

algebra to find:

$$\begin{aligned} f(A)x &= Vf(\Lambda)V^T x \\ &= Vf(\Lambda)V^T(\sum_{i=0}^{k} \alpha_i v_i) \\ &= \sum_{i=0}^{k} \alpha_i V f(\Lambda) V^T v_i \\ &= \sum_{i=0}^{k} \alpha_i V f(\Lambda) e_i \ (V^T v_i = e_i \text{ for symmetric } A. \text{ See section 1.1.1}) \\ &= \sum_{i=0}^{k} \alpha_i v_i f(\lambda_i) \end{aligned}$$

Which is easy to compute, since we only need the first k eigenvectors and eigenvalues, which we already have. We will measure the Lanczos generated approximation y with this value $\sum_{i=0}^{k} \alpha_i v_i f(\lambda_i)$

In `serial/`, please run `make numerical_test -k (krylov_dim)`, where `krylov_dim` is some number between 0 and 100.

Here the test is run for graph `data/NotreDame_yeast` (see appendix A) with 2114 rows and columns and 2440 nonzero entries, and `cond(A) = inf` as computed in MATLAB, making this a very poorly conditioned matrix. The vector x is a linear combination of the first 100 eigenvalues.

In figure 7 we can see that the Lanczos approximation converges remarkably quickly, needing only 23-30 iterations to reach a relative error of around 1e-15. Note that this error term is calculated as the norm of the vector `(analytic_ans - lanczos_ans)`. As we increase the dimension of the Krylov subspace this error does not decrease lower than 1e-15 and eventually numerical errors begin to creep in. Note that this is for a relatively small matrix $A \in \mathbb{R}^{n \times n}$, $n = 2114$. The dimensionality reduction factor is say $2114/30 \approx 70$, which demonstrates that the Lanczos method gives an extremely good approximation for very little work, even in the case where cond(A) is greater than can be represented in finite precision. This is an extremely good result.

4.4 Reorthogonalizing

In the Lanczos algorithm, each q_i is only orthogonalized against the previous q_{i-1} and q_{i-2}. This can cause problems in floating point arithmetic, as the sequence of q_js may lose orthogonality as the subspace grows [12]. Therefore, it may be feasible to reduce the relative error of the Lanczos approximation below the 1e-15 threshold if we can guarantee that the q_i remain orthogonal.

The first approach was to use an Arnoldi process every k iterations in order to reorthogonalize the q_i against all previous q_j. In `serial/` run `make numerical_test_orthog` to see sample output. As is seen in Figure 8 this did not give any better convergence than the pure Lanczos algorithm, which only orthogonalizes q_i against the previous two vectors in the basis.

Another approach was to reorthogonalize the Q matrix in its entirety once the Lanczos algorithm had terminated. This was to be done using a $Q = Q_Q R_Q$ factorization, where Q_{old}

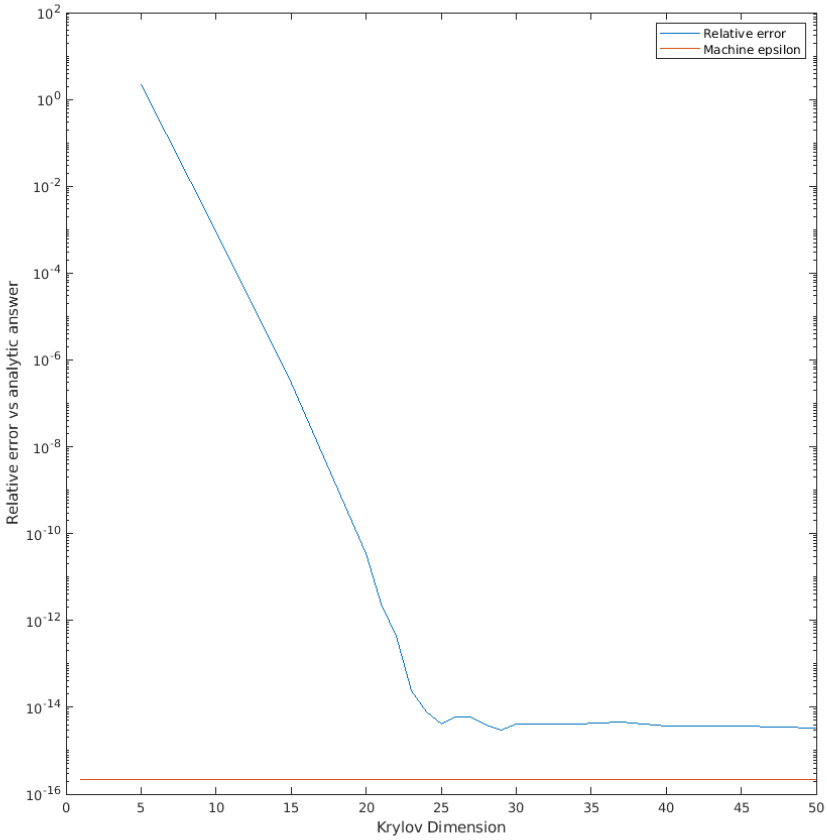
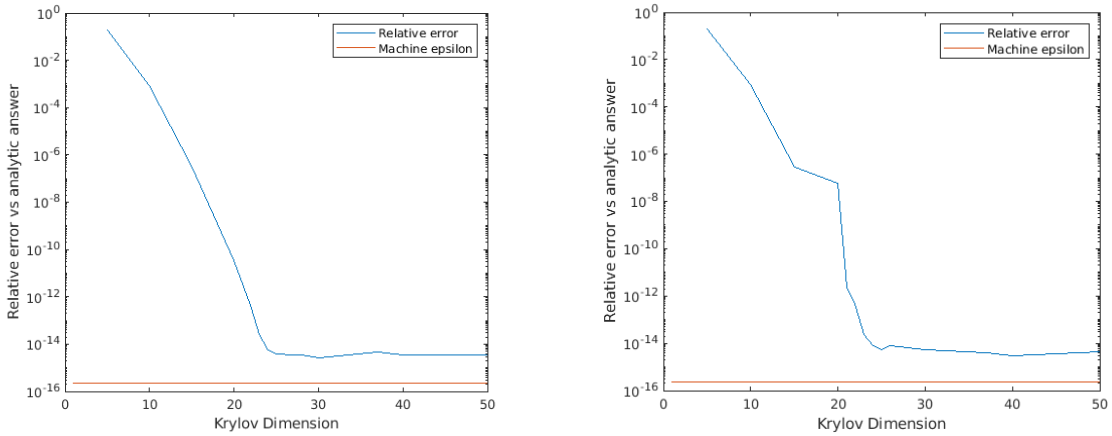


Figure 7: Checking the convergence of the Lanczos approximation for $e^A x$

would be replaced by the Q_Q found using the QR routine. However, LAPACKE routines were used for this QR factorization, which does not guarantee any particular method to constructing Q and R, so Q_Q was in fact very different from Q_{old}, meaning its correspondence with the previously generated α, β vectors was very poor. This gave very bad results. For demonstration, in `serial/tests/numerical_test_orthog.cc` change the `bool reorthogonalize` to `true`, and run `make numerical_test_orthog` from `serial/` in order to see the results of this QR reorthogonalization.

After this experimentation it became clear that the 1e-15 relative error term was about as accurate as the method could be, which was considered good enough to proceed. As [14] mentions, "Understanding when and why the Lanczos algorithm runs efficiently in the face of numerical breakdown has been the subject of intensive research for decades." The orthogonality of the basis vectors for the Lanczos approximation does not actually guarantee any greater stability or convergence than (slightly) non-orthogonal basis vectors. Therefore, the pure Lanczos algorithm is not only the easiest, but also as numerically accurate as is feasible to compute the problem in question.



(a) Reorthogonalizing every 2 iterations of Lanczos (b) Reorthogonalizing every 4 iterations of Lanczos

Figure 8: Using a full Arnoldi process every k iterations of the Lanczos algorithm to orthogonalize q_k against all previous q_i

5 Parallelization

In order to parallelize the Lanczos approximation to $e^A x$, it suffices to parallelize each individual operation that our method contains.

In order to parallelize the Lanczos method, where iterations are inherently sequential, [15] notes that the best approach is to parallelize each of the matrix and vector operations that the Lanczos operation calls. These operations are:

- Sparse matrix vector multiplication (SPMV)
- Vector dot product.
- Vector norm.
- Saxpy $(ax + y)$.

These operations are well suited to CUDA implementation.

The next operation called by the method is the eigendecomposition of the tridiagonal T. This is computed by LAPACKE in $O(k^2)$ time [14], which is negligible since k is chosen to be small as the dimension of the Krylov subspace. Since this operation takes such little time, this implementation will not try to parallelize the eigendecomposition, so it will happen on the CPU.

The next operation in the method is evaluating $QVf(\Lambda)[V^T]_1\|x\|$. It is better to evaluate this from right to left, so as to avoid the $O(n^3)$ matrix-matrix multiplication QV. If instead we work from right to left then we are left with multiple matrix-vector multiplications, which are $O(k^2)$ and $O(kn)$. The various combinations of serial and parallel execution for each of these operations will be examined in section 6.4.

6 Performance and Optimisation

In the Lanczos decomposition, sparse matrix-vector multiplication (**SPMV**) is the most time-consuming function call in each cycle of the iteration. We can see this in the output of `gprof` for the serial version of our code (Listing 2). We can see that there were 1000 calls to `spMV`, meaning that the Lanczos decomposition in this case has run for $r = 1000$ iterations.

```
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 52.25      4.03     4.03     1000     4.03     4.03  void spMV<float>(...)
 23.60      5.86     1.82                             lanczosDecomp::decompose()
 20.88      7.47     1.61  2000214     0.00     0.00  std::pair<std::_Rb_tree_iterator<Edge>, ...
  2.07      7.63     0.16        1   160.16   160.16  parseArguments(int, char**, std::__cxx11 ...
  0.52      7.67     0.04     3206     0.01     0.06  std::mersenne_twister_engine<unsigned long, ...
  0.39      7.70     0.03                             adjMatrix::random_adj()
  0.26      7.72     0.02  2000223     0.00     0.00  adjMatrix::barabasi(unsigned int)
  0.00      0.20     0.00        1     0.00     0.00  eigenDecomp::~eigenDecomp()
```

Listing 2: Flat profile of serial Lanczos Decomposition. Function names truncated for readablility.

The optimisation of the SPMV kernel will have the greatest impact on the overall runtime of the parallel Lanczos algorithm, therefore most of the programming effort will go into making this SPMV kernel as fast as possible.

However, SPMV as well as other kernels make use of reduce operations as set out in section 6.1.1. Therefore these are the kernels that we will examine first.

6.1 Linear Algebra Operations

6.1.1 Reduce Operations: Dot Product, Norm and Reduce

Please see `parallel/lib/cu_linalg.cu` for implementation.

Reduce operations are notoriously difficult to optimize in CUDA. This is due to the difficulty in sharing data among blocks, which is precisely the functionality that is required in order to reduce an array of values into a single value. It is the programmer's challenge to do so in a way that exploits the hardware to its full potential.

The operations: dot product, norm and reduce, will be considered equivalent since all consist of reducing an array of values into a single value. While the logic of each of these kernels is almost equivalent, all operations have been implemented separately. For instance, `norm_sq` could be replaced with `dot_prod` by passing in the same array for both entries. However, this would run slower than the `norm_sq` function since the kernel would be loading each entry twice, which is a waste of bandwidth. This specialized approach makes for optimised code.

The primary difficulty in reduce operations is grid synchronization, which is necessary when sharing memory between blocks. Recent versions of CUDA have introduced grid synchronization operations. While these do indeed facilitate the sharing of data among blocks, as [17] notes: "No matter how small the grid is, it seems that it [grid synchronization] is still slower than the overhead of kernel launch."

With that in mind we have a few possible ways to perform a reduction, with different approaches to grid synchronization.

1. Launch the reduction kernel on only one block. This eliminates the need for grid synchronization however is wasteful, as very little concurrency will be achieved.

2. Use global synchronization calls to ensure safe sharing of data among blocks. As [17] notes this (in the current CUDA version) performs poorly.

3. Use multiple kernel invocations. Kernels beginning and ending perform a *natural* grid sync, since the old grid is destroyed and a new one created. As [17] notes this overhead is the fastest grid synchronization method available.

Therefore we will proceed with method 3. See 9. The first kernel calls each block to reduce its part of the array into a single value, which is then written some `tmp_d` array in global memory (which has size `num_blocks_in_grid`). Then the second kernel is launched on a single block, and reduces the `tmp_d` array into a single value. It is safe for the single block to access `tmp_d` since the second kernel will only be launched once all blocks have returned from the first kernel.

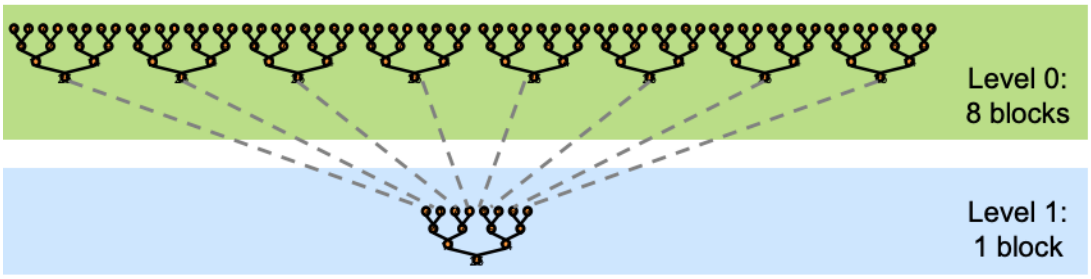


Figure 9: Two reduce kernel invocations. Source [2]

Note that the same reduce operation can be called for both invocations. In the manner of:

```
cu_reduce<<<number_blocks_in_grid, BLOCKSIZE>>>(x_d, n, tmp_d);
cu_reduce<<<1, BLOCKSIZE>>>(tmp_d, number_of_blocks_in_grid, ans_d);
```

Listing 3: Calling the same kernel twice in order to achieve grid synchronization

The answer of the reduction will be stored at `ans_d[0]`.

Now to examine the entire reduction kernel. The kernel draws heavily on [2]:

```
template <typename T, unsigned blockSize>
__global__ void cu_reduce(T * a, const unsigned n, T * ans) {
    unsigned tid = threadIdx.x;
    int i = blockIdx.x*blockSize*2 + tid;
    unsigned gridSize = blockSize*2*gridDim.x;

    __shared__ T sdata[2*blockSize];
    sdata[tid]=0.0; sdata[tid+blockSize]=0.0;

    // Getting partial sums into shared memory
    while (i+blockSize<n) {
        sdata[tid] += a[i]+a[i+blockSize];
        i += gridSize;
    }
```

```
15      if (i<n) sdata[tid] += a[i];
16
17      __syncthreads();
18
19      // Folding the partial sums together
20      if (blockSize == 1024) {
21          (if tid < 512) sdata[tid] += sdata[tid+512]; __syncthreads(); }
22      if (blockSize >= 512) {
23          (if tid < 256) sdata[tid] += sdata[tid+256]; __syncthreads(); }
24      if (blockSize >= 256) {
25          (if tid < 128) sdata[tid] += sdata[tid+128]; __syncthreads(); }
26      if (blockSize >= 128) {
27          (if tid < 64) sdata[tid] += sdata[tid+64]; __syncthreads(); }
28      if (blockSize >= 64) {
29          (if tid < 32) sdata[tid] += sdata[tid+32]; __syncthreads(); }
30
31      if (tid < 32) sdata[tid] = warpReduceSum<T,blockSize>(sdata, tid);
32      if (tid == 0) ans[blockIdx.x] = sdata[0];
33  }
```

Listing 4: An optimized CUDA reduce kernel

In this kernel all threads race through the entire array `a` and store the partial sums in shared memory. This loading of `tmp_s` in a single sweep maximizes the coalescence of memory access. Then each block condenses all of its values in a folding pattern into a single value, which is then written to global memory in the final line.

[2] notes that the double load in line 12 gains a marginal speedup. The `warpReduceSum` (Listing 12) device function, allows entire warps to *clock off* and exit, instead of going through all of the logic hidden within the function, which is required only for threads 0-31. This frees up space in the streaming multiprocessors, meaning kernels can achieve higher occupancy.

BlockSize is used as a templated parameter in order to streamline machine code. See section 6.5 for a full description.

Speedups and Choosing Blocksizes. Now to compare the speedups for various n values and different blocksizes.

Since our reduction operations consist of two kernel invocations (Listing 3), we must first find the ideal blocksize for a single-block grid performing the second reduction, on a small array. Then we can work backwards to find the best blocksize for the first kernel, given that the second kernel is assigned the best blocksize. Note that the dimension of the `tmp_d` array being reduced in the second kernel invocation is likely to have size $\approx 10^3$.

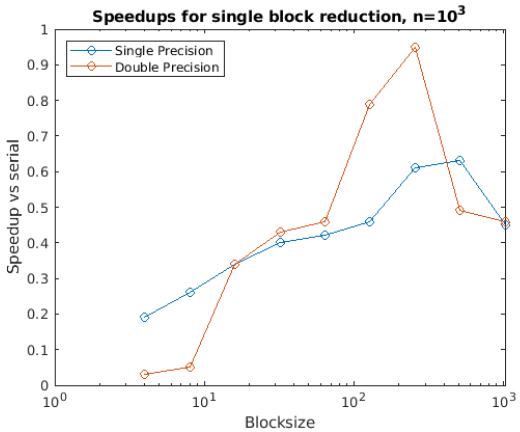


Figure 10: Speedups for a single block reduction, $n = 10^3$.

In Figure 10 we see that CUDA reduction is slower than serial for $n = 10^3$, however this second kernel execution makes up a fraction of the time of the overall reduction. Moreover we are more than happy to suffer a slight slowdown if it eliminates the need for memory transfers, which would certainly throttle performance in such a fast-executing operation. Therefore 256 is an ideal blocksize for the second reduction kernel, for all reduce operations.

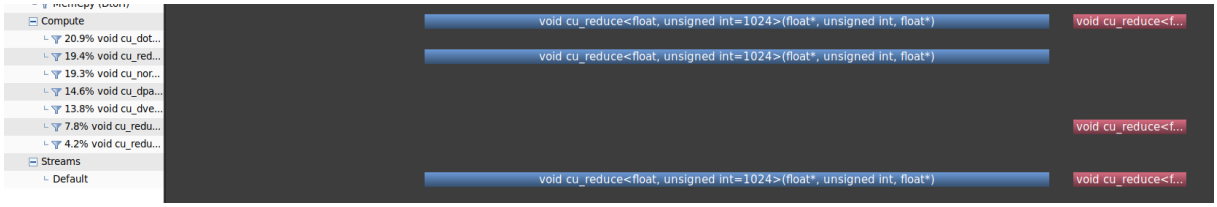


Figure 11: NVVP timeline of dual kernel launch. The second reduction kernel makes up a small portion of the dual kernel operation.

From figures 12 -14 It's clear that CUDA only offers significant speedups for $n > 10^5$. However, even for $n \approx 10^5$ there are considerable speedups vs CBLAS with automatic multi-threading. Moreover, the ability to perform all Lanczos operations *on card* allows for very low latency between operations. Meaning it is beneficial to run all operations on card, even if serial and CUDA execution times were to be identical, since it removes the need for `memcpy` calls.

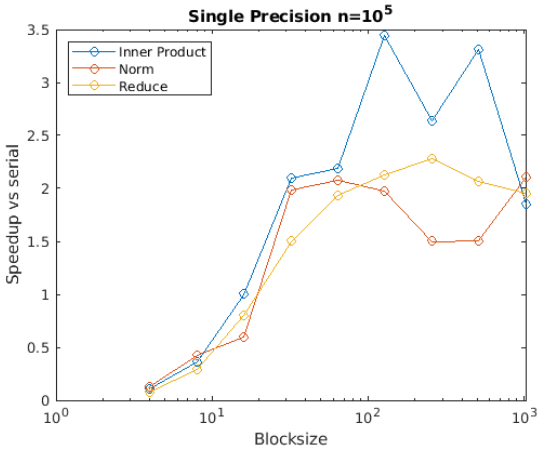
Unfortunately, since we are using `blockSize` as a templated parameter, the call to each reduce operation needs to be resolved statically. This means we cannot implement some kind of logic such as `if (n > 100000) blockSize=256`, since the templated function needs to be chosen for `blockSize` by the compiler. Also we are not going to make `n` a `constexpr`, since we want the ability to choose a matrix to compute on on the fly. Therefore, it is necessary to choose a predetermined blocksize for each operation regardless of the size of n.

Examining 12 -14 it is clear that blocksizes 128, 256 are almost universally good. It seems sensible to prioritize higher performance for smaller n values, since all block sizes in the range 128-512 give good performance for large n values. Poor performance for small n will be more detrimental to performance than marginally suboptimal performance for large n.

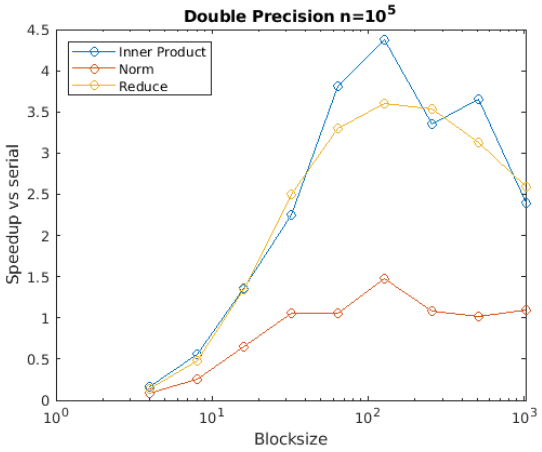
	Inner Product	Norm	Reduce
Best block size	128	128	256

6.1.2 Warp-Level Primitives

```
template <typename T, unsigned blockSize>
__inline__ __device__ T warpReduceSum(T val) {
  if (blockSize >= 32) val += __shfl_down_sync(FULL_MASK, val, 16);
  if (blockSize >= 16) val += __shfl_down_sync(FULL_MASK, val,  8);
  if (blockSize >=  8) val += __shfl_down_sync(FULL_MASK, val,  4);
  if (blockSize >=  4) val += __shfl_down_sync(FULL_MASK, val,  2);
  if (blockSize >=  2) val += __shfl_down_sync(FULL_MASK, val,  1);
```

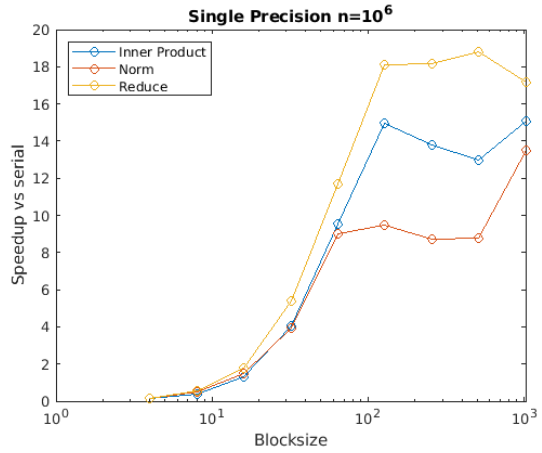


(a) Single precision $n = 10^5$

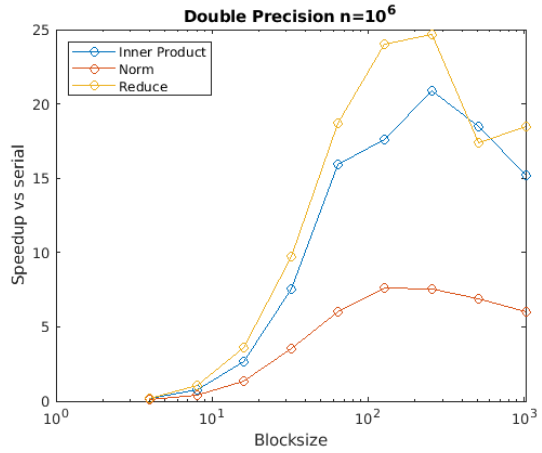


(b) Double precision $n = 10^5$

Figure 12: $n = 10^5$: Speedups for blocksizes: 4, 8, 16, 32, 64, 128, 256, 512, 1024.

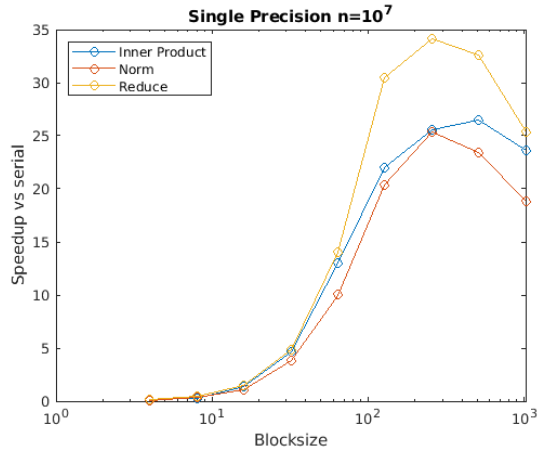


(a) Single precision $n = 10^6$

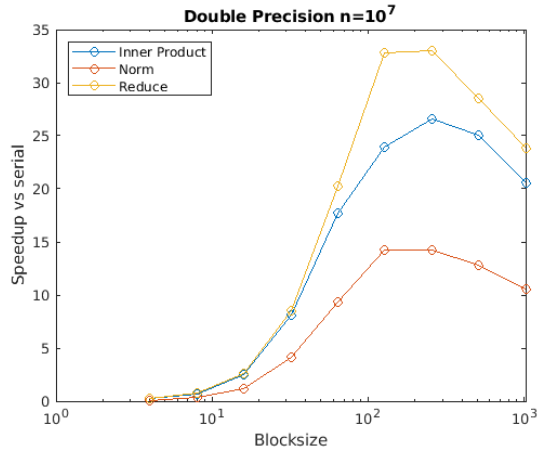


(b) Double precision $n = 10^6$

Figure 13: $n = 10^6$: Speedups for blocksizes: 4, 8, 16, 32, 64, 128, 256, 512, 1024.



(a) Single precision $n = 10^7$



(b) Double precision $n = 10^7$

Figure 14: $n = 10^7$: Speedups for blocksizes: 4, 8, 16, 32, 64, 128, 256, 512, 1024.

```
8   return val;
9 }
```

Listing 5: Warp reduce using warp-level primitives

The final step of the reduction is to call `warpReduceSum()`, which calls the warp-level primitive `__shfl_down_sync()` in order to condense the warp values into a single value. As [18] notes:

"1. *Shuffle replaces a multi-instruction shared memory sequence with a single instruction, increasing effective bandwidth and decreasing latency.*

2. *Shuffle does not use any shared memory.*

3. *Synchronization is within a warp and is implicit in the instruction, so there is no need to synchronize the whole thread block with* `__syncthreads()`."

Therefore this device function ought to run faster than the `warpReduce` seen in section 6.5. However, examining the machine code in Listings 6 and 7 shows that the shared memory reduction actually contains fewer instructions. But we trust Nvidia [18] in their assertion that the warp level directives are faster than shared memory. Meaning that perhaps each instruction takes fewer cycles for the warp level reduction (Listing 7) than for the shared memory reduction (Listing 6).

However, this function will only be called once per kernel and it is a very short device function, meaning that its impact on the execution speed of the surrounding reduce kernels will be almost imperceptible. Nevertheless the final implementation shall include the warp level directives version of the warp reduce.

```
shl.b32          %r25, %r1, 3;
mov.u32          %r26, hello()::sdata;
add.s32          %r27, %r26, %r25;
ld.volatile.shared.f64  %fd11, [%r27];
ld.volatile.shared.f64  %fd12, [%r27+128];
add.f64          %fd13, %fd12, %fd11;
st.volatile.shared.f64  [%r27], %fd13;
ld.volatile.shared.f64  %fd14, [%r27];
ld.volatile.shared.f64  %fd15, [%r27+64];
add.f64          %fd16, %fd15, %fd14;
st.volatile.shared.f64  [%r27], %fd16;
ld.volatile.shared.f64  %fd17, [%r27];
ld.volatile.shared.f64  %fd18, [%r27+32];
add.f64          %fd19, %fd18, %fd17;
st.volatile.shared.f64  [%r27], %fd19;
mov.u32          %r28, 16;
ld.volatile.shared.f64  %fd20, [%r27];
ld.volatile.shared.f64  %fd21, [%r27+16];
add.f64          %fd22, %fd21, %fd20;
st.volatile.shared.f64  [%r27], %fd22;
mov.u32          %r29, 8;
ld.volatile.shared.f64  %fd23, [%r27];
ld.volatile.shared.f64  %fd24, [%r27+8];
add.f64          %fd1, %fd24, %fd23;
st.volatile.shared.f64  [%r27], %fd1;
```

Listing 6: `warpReduce1()`: A shared memory reduction (see section 6.5 for definition)

```
mov.b64 {%r5,%r6}, %fd1;
mov.u32          %r30, 2;
mov.u32          %r31, 31;
mov.u32          %r32, 16777215;
shfl.sync.down.b32  %r8|%p2, %r6, ...;
shfl.sync.down.b32  %r7|%p3, %r5, ...;
mov.b64 %fd2, {%r7,%r8};
add.f64          %fd3, %fd1, %fd2;
mov.b64 {%r9,%r10}, %fd3;
shfl.sync.down.b32  %r12|%p4, %r10, ...;
shfl.sync.down.b32  %r11|%p5, %r9, ...;
mov.b64 %fd4, {%r11,%r12};
add.f64          %fd5, %fd3, %fd4;
mov.b64 {%r13,%r14}, %fd5;
mov.u32          %r33, 4;
shfl.sync.down.b32  %r16|%p6, %r14, ...;
shfl.sync.down.b32  %r15|%p7, %r13, ...;
mov.b64 %fd6, {%r15,%r16};
add.f64          %fd7, %fd5, %fd6;
mov.b64 {%r17,%r18}, %fd7;
shfl.sync.down.b32  %r20|%p8, %r18, ...;
shfl.sync.down.b32  %r19|%p9, %r17, ...;
mov.b64 %fd8, {%r19,%r20};
add.f64          %fd9, %fd7, %fd8;
mov.b64 {%r21,%r22}, %fd9;
mov.u32          %r34, 1;
shfl.sync.down.b32 %r24|%p10, %r22, ...;
shfl.sync.down.b32 %r23|%p11, %r21, ...;
mov.b64 %fd10, {%r23,%r24};
add.f64          %fd25, %fd9, %fd10;
```

Listing 7: `warpReduceSum()`: a reduction using warp-level directives

6.1.3 Saxpy

See `lib/cu_linalg.cu` for implementation.

Saxpy (single precision $ax + y$) and daxpy (double precision $ax + y$) are some of the easiest operations to implement in CUDA. It is a trivially parallel operation so there is no need to make use of the shared memory or any advanced CUDA technique. The Lanczos method calls for the operation:

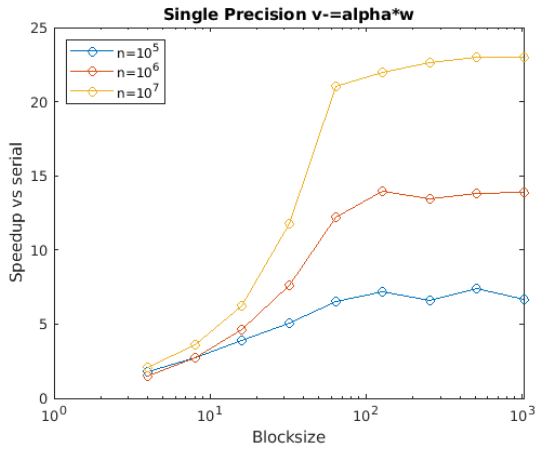
$$v = v - \alpha * w \tag{19}$$

As well as the operation:

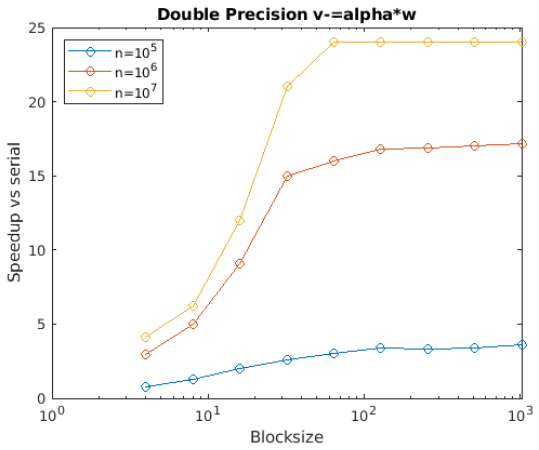
$$v = v/\beta \tag{20}$$

To minimize the data requirements of equation 20, separate kernels are written for each of these operations, as is the case for the reduce operations.

From figures 15, 16 we see that these trivially parallel operations have little preference for blocksize. 256 will be chosen as an ideal blocksize for both saxpy operations.

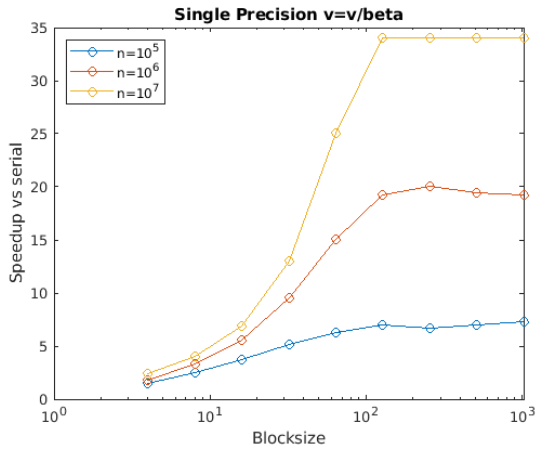


(a) Single precision $v = v - \alpha * w$

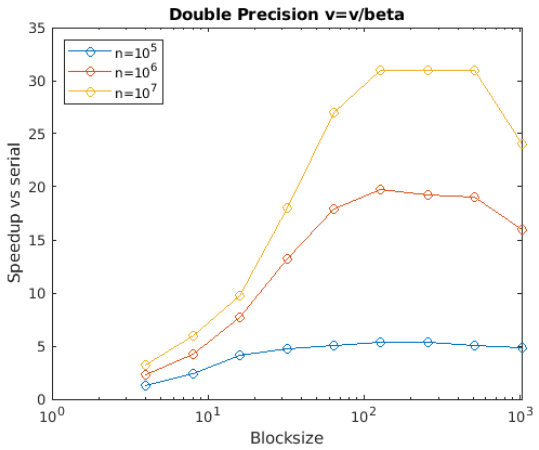


(b) Double precision $v = v - \alpha * w$

Figure 15: $v = v - \alpha * w$ for blocksize 4, 8, 16, 32, 64 128, 256, 512, 1024



(a) Single precision $v = v/\beta$



(b) Double precision $v = v/\beta$

Figure 16: $v = v/\beta$ for blocksize 4, 8, 16, 32, 64 128, 256, 512, 1024

6.2 SPMV

As mentioned in section 6, SPMV is the most time consuming operation (cumulatively) in the Lanczos approximation. Therefore we will use many different approaches to attack this problem.

6.2.1 Basic SPMV

The basic structure of the serial algorithm is as follows:

```
void spMV(const adjMatrix & A, const T * const in, T * const out) {
  for (auto i = 0u; i < A.n; ++i) {
    out[i] = 0.0;
  }
  for (auto i = 0u; i < A.n; ++i) {
    for (auto j=A.IA[i];j<A.IA[i+1];j++)
      out[i] += in[A.JA[j]];
  }
```

```
}
```

Listing 8: Serial C++ code for SPMV

Where `IA`, and `JA` are as outlined in section 3.1. It is clear that the function runs in $O(|E|)$ time, since `row_offset[n] = |E|`. This relatively simple function belies the difficulty of parallelizing this code. This is due to the fact that A, being sparse, is liable to take any shape:

- A may be well balanced, where the degree distribution of each node has very little variance. Balanced graphs may represent stencils of objects, and have predictable structure.
- A may be irregular, where the degree distribution for nodes has high variance. Irregular networks occur frequently in real-world scenarios; it is natural that in social or financial networks, some nodes have a drastically higher degree than others.

While parallelizing SPMV for the former, balanced case is relatively easy, parallelization for irregular graphs will be the main computational challenge in this CUDA Lanczos decomposition.

```
// 1 thread is responsible for 1 row of A
__global__ void cu_spMV1(U *const IA,  U *const JA, const U n, T *const x, T *
    ans)
{
  auto tid {blockDim.x * blockIdx.x + threadIdx.x};
  if (tid < n) {
    T t_ans{0};
    for (auto i = IA[tid]; i < IA[tid + 1]; i++)
      t_ans += x[JA[i]];
    ans[tid] = t_ans;
  }
}
```

Listing 9: cu_SPMV1, a naive CUDA kernel for SPMV

The first step is to transfer the logic wholesale into CUDA format. This results in a very simple kernel, with relatively good performance. We only need to load a handful of values from global memory, namely `IA[tid]`, which allows coalesced memory access among threads, and then each cycle access `JA[i]`, which will not allow coalesced memory access among threads, since thread k will be accessing `JA[IA[tid]+count]` whereas thread $k+1$ will be accessing `JA[IA[tid+1]+count]`, which are liable to be far from each other. `x` will be accessed in a similarly non-coalesced fashion, which can result in poor performance.

However, the main problem with this kernel is that it does not attempt to balance the loads among threads. A pathological example that would make this kernel perform very poorly would be the case where a single node is connected to every other node, meaning one single thread will have to do $O(|E|)$ work, as opposed to an ideally balanced $O\left(|E|\right) \#$ num threads . For this reason, a concerted effort at load balancing is required.

6.2.2 Load Balanced SPMV

There are a number of ways to spread load more evenly among threads.

Dynamic Number of Rows Per Block One approach is to assign a dynamic number of rows to each block. This method was adapted from an 'Improved PCSR' method [16], with improvements obtained by using block reduce operations instead of single thread reductions as in the original algorithm. The entire matrix will be partitioned into `num_blocks` blocks, where each block is responsible for as many rows it can fit into shared memory. If a single row is too big to fit into shared memory then said row will be operated on by a single block. This addresses the problem of load balancing, since each block is guaranteed to have close enough to `SHARED_SIZE/sizeof(T)` elements to compute the row sum for. However, this requires an additional array `blockrows` so that each block knows which of the rows it is to compute on.

```
template <typename T, typename U, unsigned blockSize>
__global__ void cu_spMV2(U *const IA, U *const JA, U *const blockrows, const U n
    , T *const x, T *ans)
{
  __shared__ T tmp_s[SHARED_BYTES/sizeof(T)];
  auto tid{threadIdx.x}, bid{blockIdx.x};
  auto startrow{blockrows[bid]}, endrow{blockrows[bid + 1]};
  auto nnz{IA[endrow] - IA[startrow]};

  auto firstcol {IA[startrow]};

  // A single long row for a whole block
  if (endrow-startrow == 1) {

    tmp_s[tid] = 0;
    tmp_s[tid+blockSize] = 0;

    for (auto i=firstcol+tid; i<IA[endrow];i+=blockSize)
      tmp_s[tid] += x[JA[i]];

    __syncthreads();

    // Reduction
    if (blockSize == 1024) { tmp_s[tid] += tmp_s[tid+512]; __syncthreads(); }
    if (blockSize >= 512) { tmp_s[tid] += tmp_s[tid+256]; __syncthreads(); }
    if (blockSize >= 256) { tmp_s[tid] += tmp_s[tid+128]; __syncthreads(); }
    if (blockSize >= 128) { tmp_s[tid] += tmp_s[tid+64]; __syncthreads(); }

    if (tid < 32) warpReduce<T,blockSize>(tmp_s, tid);
    if (tid == 0) ans[startrow] = tmp_s[0];
    return;
  }

  assert(nnz <= (SHARED_BYTES/sizeof(T)));

  // Load all rows into shared memory
  for (auto i = firstcol+tid; i < IA[endrow]; i += blockDim.x)
    tmp_s[i-firstcol] = x[JA[i]];

  __syncthreads();
```

```
40
41    // Now reduce the rows using one thread per row
42    // This scalar reduction is faster than a multiple reduction
43    // as used in cu_linalg functions, since row_e - row_s is small
44    auto num_rows{endrow - startrow};
45    if (tid < num_rows && startrow+tid < n)
46    {
47      auto sum{0.0};
48      auto row_s{IA[startrow + tid] - firstcol};
49      auto row_e{IA[startrow + tid + 1] - firstcol};
50      for (auto i = row_s; i < row_e; i++)
51        sum += tmp_s[i];
52      ans[startrow + tid] = sum;
53    }
54  }
```

Listing 10: SPMV with dynamic number of rows per block

As mentioned in code comments, when a block is assigned multiple rows the maximum size of a given row will be `49152/sizeof(T)` (49152 bytes is the size of shared memory). From figure 10 we see that multiple-style reductions are not efficient for this size n, meaning a scalar reduction, where multiple threads per block may be idle, is more efficient. Paradoxically it is more efficient for multiple threads per block to be inactive, than for all to have to partake and coordinate in a small reduction.

Dynamic Parallelism While `SPMV2` does balance load more effectively among blocks, it also increases the memory requirements of the kernel, since the array `blockrows` is required to tell each block which rows they are responsible for. Since SPMV is a memory bound kernel this will decrease performance (which will hopefully be more than offset by better load-balancing). It is worth testing if we can get load-balanced SPMV but without the need for extra data.

A possible approach would be to launch a naive `SPMV1` style kernel, where each thread is responsible for a single entry in the output vector. However, unlike in `SPMV1`, we can use dynamic parallelism to launch extra blocks from threads with very long rows to compute on. This eliminates the need for extra data, however it may cause some deadlocking if some threads in a block are waiting for a child kernel to finish. This would be especially problematic if the inactive threads are waiting on a neighbouring thread's kernel to return, making them unable to compute on their own row. Nevertheless we will compare performance at the end of this section.

```
1  template <typename T, typename U, unsigned blockSize>
2  __global__ void cu_spMV3_child(U * const JA, const U nnz, T * const x, T * const
       ans) {
3    __shared__ T tmp_s[blockSize*2];
4    auto tid {threadIdx.x};
5    tmp_s[tid] = 0;
6    for (auto i=tid;i<nnz;i+=blockDim.x) tmp_s[tid] += x[JA[i]];
7
8    if (tid < 32) warpReduce<T,blockSize>(tmp_s, tid);
9    if (tid == 0) ans[0] = tmp_s[0];
```

```
}

template <typename T, typename U, unsigned blockSize>
__global__ void cu_spMV3(U * const IA, U * const JA, const U n, const U avg_nnz,
    T * const x, T * const ans) {
  auto tid {blockDim.x*blockIdx.x+threadIdx.x};
  if (tid >= n) return;
  auto start_of_row {IA[tid]};
  auto end_of_row {IA[tid+1]};
  auto nnz {end_of_row-start_of_row};

  if (nnz / THRESHOLD >= avg_nnz) {
    cu_spMV3_child<T,U, CHILD_BLOCK_SIZE><<<1,CHILD_BLOCK_SIZE>>>
        (&JA[start_of_row], nnz, x, &ans[tid]);
  } else {
    T t_ans{0};
    for (auto i = IA[tid]; i < IA[tid + 1]; i++)
      t_ans += x[JA[i]];
    ans[tid] = t_ans;
  }
}
```

Listing 11: Dynamically parallel SPMV kernel

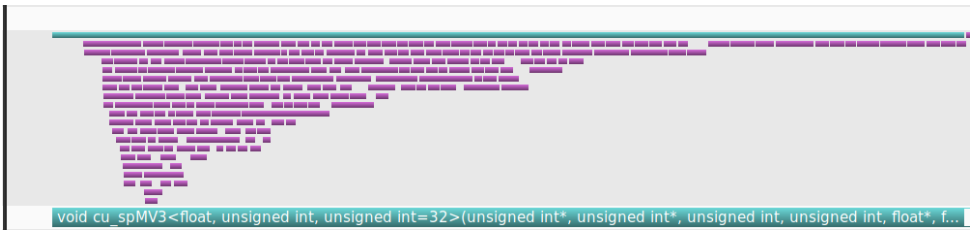


Figure 17: `SPMV3` (green) making many calls to `SPMV3_child` (purple), corresponding to a low `THRESHOLD` value (`THRESHOLD=50`). Here the execution of the entire kernel is held up by the calls to `SPMV3_child`. Visualized with Nvidia Visual Profiler

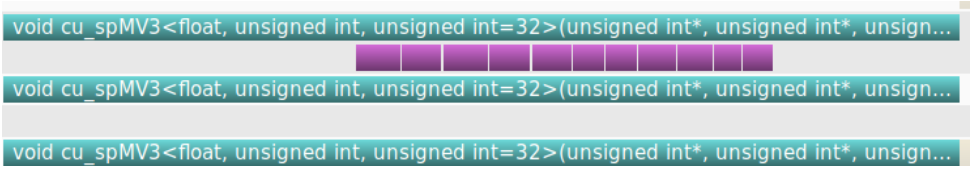


Figure 18: Lesser dynamic parallelism in this call to SPMV3, corresponding to a higher `THRESHOLD` value (`THRESHOLD=200`). Note that kernel execution is not stalled by calls to `SPMV3_child`.

For this kernel, not only do we need to find the ideal `blockSize`, but also the ideal `THRESHOLD` value, as well as the ideal `CHILD_BLOCK_SIZE`. If we optimise the `THRESHOLD` value well enough, then we are guaranteed to get a kernel at least *nearly* as fast as the naïve `cu_SPMV1`, (in the worst case scenario that the dynamic parallelism *only* pessimizes execution time, the value

`THRESHOLD=Inf` will make `SPMV3` almost identical to `SPMV1`, as it omits the call to `SPMV3_child`). However, we suspect that for a large enough `THRESHOLD` value, we will obtain a better speedup than `SPMV1`.

The child kernel calls will only occur in the event that A is highly imbalanced, so `SPMV3` may be understood as "`SPMV1` with an insurance clause": something to resort to if A is highly imbalanced. The price of this insurance clause is the added instructions in lines 16-20 of `SPMV3`. Since the general philosophy of optimisation seeks to get *good* performance *on average* rather than *amazing* performance *for particular cases*, this tradeoff may be considered worthwhile.

Now to test `cu_SPMV3` against `cu_SPMV1` for varying `THRESHOLD` values. The tests will be performed without loss of generality on a highly imbalanced matrix, the self generated Barabasi (see appendix A).

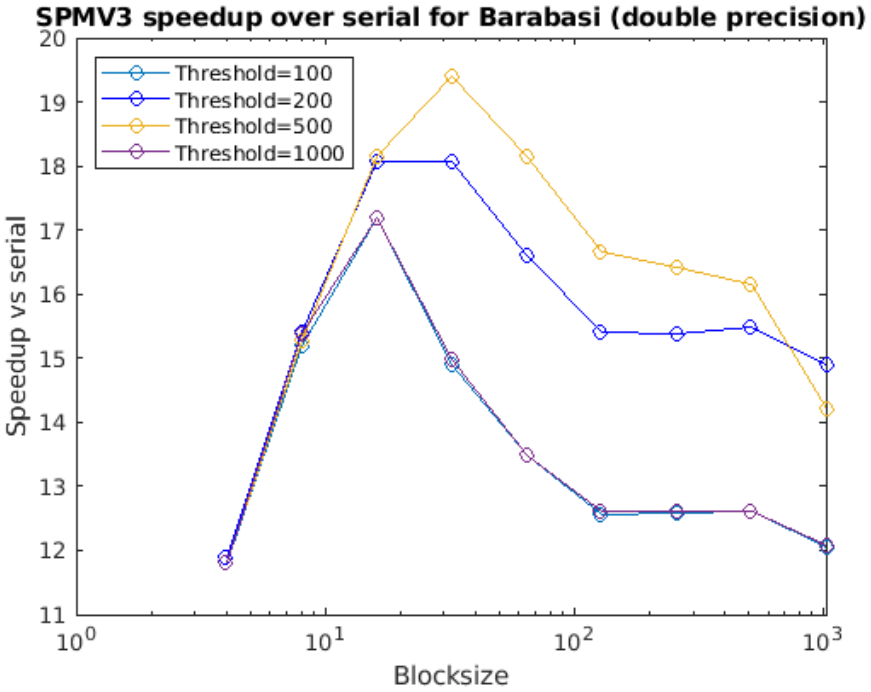


Figure 19: Speedup of `cu_SPMV3` against serial execution speed.

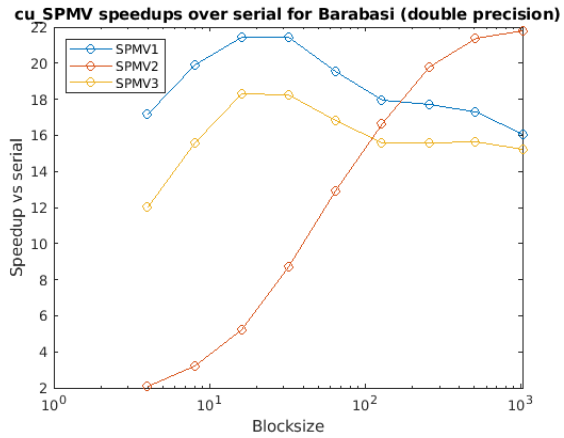
From Figure 19 we can see that `THRESHOLD=500` is about as good as can be expected for the dynamically parallel kernel. The ideal blocksize for the child kernel was found to be the size of a single warp, 32 (illustration omitted).

6.2.3 Comparing Performance of SPMV kernels

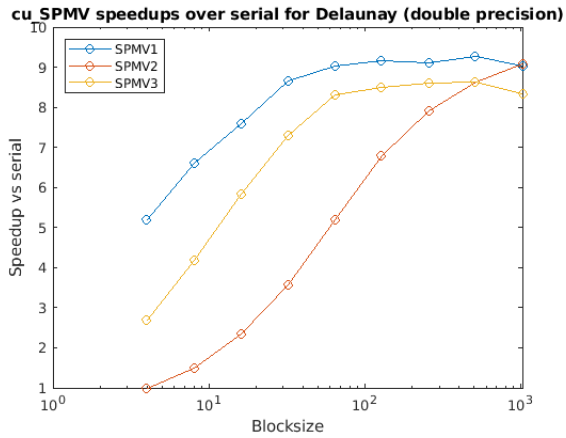
Now to compare the performance of `cu_SPMV1`, `cu_SPMV2` and `cu_SPMV3` with respect to different blocksizes.

NVVP Reports

SPMV1 performs remarkably well for such a simple kernel. Slightly concerning is the difficulty in choosing a blocksize, especially if the matrix is imbalanced, as in Figure 20a. However, the

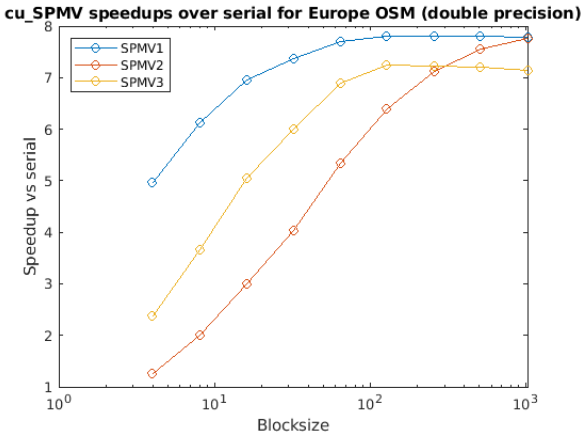


(a) SPMV speedups for Barabasi matrix.

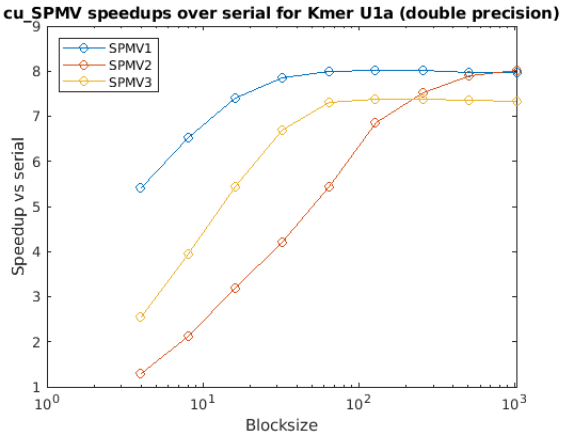


(b) SPMV speedups for delaunayn24 matrix.

Figure 20: SPMV speedups.



(a) SPMV speedups for europe osm matrix.



(b) SPMV speedups for Kmer U1a matrix.

Figure 21: SPMV Speedups

kernel runs fast and achieves very good occupancy for very little effort.

SPMV2 The performance of SPMV2 for a given blocksize does not vary significantly for differently shaped inputs. This makes it easy to choose the optimal blocksize 1024. It is clear that the load balancing of SPMV2 has managed to offset the cost of extra data transfer, but it does not significantly outperform SPMV1, despite the very good occupancy of 99.8%. However, the predictability of the performance of SPMV2 with respect to blocksize makes it a better SPMV candidate than SPMV1.

SPMV3 It is disappointing that the dynamically parallel SPMV3 fails to outperform SPMV1, despite using the optimal parameters for `THRESHOLD` as well as `CHILD_BLOCK_SIZE`. The supposed benefit of load balancing has not given any performance benefits. This is most likely due to warp divergence, indicated by the very low occupancy that the kernel achieves. This presumably occurs as child kernel calls cause entire blocks to stall while waiting on the child process to

	SPMV1	SPMV2	SPMV3
Theoretical Occupancy	100%	100%	50%
Achieved Occupancy	83%	99.8%	25.1%

Table 3: Statistics taken from NVVP. See `code/reports/`.

return.

Conclusion The Lanczos algorithm will use SPMV2 with the ideal blocksize 1024. This is due to its predictable performance for blocksize = 1024 for a range of input matrices.

Further Approaches: In `parallel/SPMV_test.cu` you may also see a call to hybrid SPMV 1 & 4. This kernel seeks to combine the dynamic approach of SMV3 but while keeping the good behaviour of SPMV1. It would do that by splitting the SPMV up into two parallel kernels.

- SPMVhA would use entire blocks to compute a given row.
- SPMVhB would use a single thread to compute a given row, as in SPMV1.

The hybrid approach relies on a preprocessing step—a permutation to put the k most connected nodes at the beginning of the matrix, and leave the remaining, smaller $n - k$ nodes at the end of the matrix. SPMVhA and SPMVhB would then be launched in parallel using CUDA streams, where SPMVhA is computing on the first k long rows, and SPMVhB is computing on the $n - k$ short rows. Note that this would not require a full sorting of nodes but just a SPMV multiplication to compute the degrees ($A\vec{1}$) and then a single sweep to see which nodes have degree greater than some threshold. More research is required in this direction. Ideally it would achieve good occupancy as well as load balancing, and would not require the extra data that SPMV2 does.

6.3 Eigendecomposition

In the serial profile (Listing 2) we see that eigendecomposition comprises less than 0.01% of running time. This is a very cheap operation since the eigendecomposition of a tridiagonal is completed in $O(r^2)$ operations, and r, the Krylov dimension, is chosen to be small. Therefore, there is no benefit in parallelizing this operation.

6.4 Multiply Out

The final process in order to generate the answer vector $y \approx f(A)x$, is to multiply the expression:

$$QVf(\Lambda)[V^T]_1\|x\|$$

As mentioned in section 5, it is better to evaluate this from right to left, so that we have two matrix-vector multiplications rather than a matrix-matrix and matrix vector multiplication. This improves time complexity from $O(k^2n)$ to $O(kn)$, a big improvement.

Unlike SPMV, which poses many difficulties with parallelization (most notably for load balancing), dense matrix-vector multiplication is trivially parallel. Therefore this section will not attempt to improve upon library code, and instead will test the performance of `cu_blas` (run on the GPU) vs. `cblas` (run with multithreading on the CPU).

See `parallel-mult-on-card/` and run `make lanczos test` to see how the performance of `cu_blas` and `cblas` compare.

Matrix	Barabasi	Delaunay	kmerU1a
Speedup vs Serial for Multiply Out	0.15	0.018	0.94

We can clearly see that multOut is not an operation that `cuBlas` is suited to, at least until n is very large. Therefore it is better to use the OpenBLAS implementation, and use the inbuilt threading of OpenBLAS to parallelize the operation as much as possible.

Using the command `OPENBLAS_NUM_THREADS=k ./lanczos_test` we will see how the OpenBLAS multiply out routine scales with added threads.

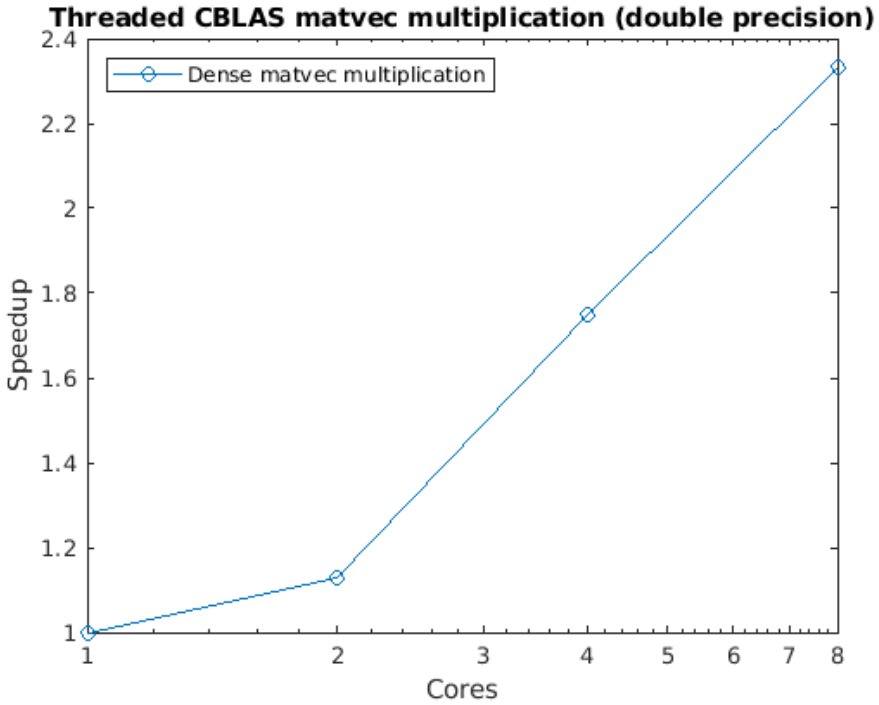


Figure 22: Multithreading CBLAS. CBLAS scales nicely with added cores.

The final implementation will use multithreaded CBLAS to perform the multOut operation, rather than cuBlas.

6.5 Templates

Templates are used extensively throughout the code. This has two purposes:

- The ease of transferring between `double` and `float` implementations of the Lanczos decomposition and other computations.

- The streamlining of compiled code due to template-generated parameters. Specifically the parameter `blockSize`.

The first point is evident. The second point is perhaps less so. This optimisation relies on the fact that `blockSize` does not need to be a dynamic variable, since it should only take on values 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024. This means that the compiler can generate all the necessary functions at compile time, which allows all loops to be completely unrolled, and eliminates the need for any `if {...} else` checks at run time. These checks are pre-resolved in each template instantiation.

To demonstrate, take two instances of a warpReduce function, taken from Mark Harris's CUDA reduction tutorial [2]. This is a function to reduce the shared memory of a warp into a single value at `sdata[0]`. One instance uses `blockSize` as a templated parameter (Listing 12), whereas the other uses `blockSize` as a dynamic parameter (Listing 13).

Note that since `warpReduce(...)` is a `__device__` function, the assembly code generated would be simply inlined into any calling `__global__` function.

```
1 template <unsigned int blockSize>
2 __device__ void warpReduce1(volatile double * sdata, const unsigned tid) {
3     if (blockSize >= 64) sdata[tid] += sdata[tid+32];
4     if (blockSize >= 32) sdata[tid] += sdata[tid+16];
5     if (blockSize >= 16) sdata[tid] += sdata[tid+8];
6     if (blockSize >= 8) sdata[tid] += sdata[tid+4];
7     if (blockSize >= 4) sdata[tid] += sdata[tid+2];
8     if (blockSize >= 2) sdata[tid] += sdata[tid+1];
9 }
```

Listing 12: `warpReduce1`: using `blockSize` as templated parameter

```
1 __device__ void warpReduce2(volatile double * sdata, const unsigned tid, const
    unsigned blockSize) {
2     if (blockSize >= 64) sdata[tid] += sdata[tid+32];
3     if (blockSize >= 32) sdata[tid] += sdata[tid+16];
4     if (blockSize >= 16) sdata[tid] += sdata[tid+8];
5     if (blockSize >= 8) sdata[tid] += sdata[tid+4];
6     if (blockSize >= 4) sdata[tid] += sdata[tid+2];
7     if (blockSize >= 2) sdata[tid] += sdata[tid+1];
8 }
```

Listing 13: `warpReduce2`: using `blockSize` as a dynamic parameter

Now to inspect the assembly code. It is clear that Listing 14 contains far fewer instructions than Listing 15. The templated version (Listing 14) omits the `if... else` statements in Listing 15 (these checks occur in the lines `setp.lt.u32 ...`), and also omits line 3 of Listing 12, since for the instantiation `blockSize=32` the result of this check is predetermined, meaning the compiler can excise the unnecessary line. For further demonstration of compiler excision this please see the instantiation `warpReduce<2>` (Listing 16). Where the code is reduced to only four instructions.

```
ld.volatile.shared.f64  %fd10, [%r4];
ld.volatile.shared.f64  %fd11, [%r4+128];
add.f64         %fd12, %fd11, %fd10;
st.volatile.shared.f64  [%r4], %fd12;
ld.volatile.shared.f64  %fd13, [%r4];
ld.volatile.shared.f64  %fd14, [%r4+64];
add.f64         %fd15, %fd14, %fd13;
st.volatile.shared.f64  [%r4], %fd15;
ld.volatile.shared.f64  %fd16, [%r4];
ld.volatile.shared.f64  %fd17, [%r4+32];
add.f64         %fd18, %fd17, %fd16;
st.volatile.shared.f64  [%r4], %fd18;
ld.volatile.shared.f64  %fd19, [%r4];
ld.volatile.shared.f64  %fd20, [%r4+16];
add.f64         %fd21, %fd20, %fd19;
st.volatile.shared.f64  [%r4], %fd21;
ld.volatile.shared.f64  %fd22, [%r4];
ld.volatile.shared.f64  %fd23, [%r4+8];
add.f64         %fd24, %fd23, %fd22;
st.volatile.shared.f64  [%r4], %fd24;

```

Listing 14: `warpReduce1<32>()` assembly code

```
setp.lt.u32     %p10, %r11, 64;
@%p10 bra       $L__BB0_19;
ld.volatile.shared.f64  %fd19, [%r4];
ld.volatile.shared.f64  %fd20, [%r4+256];
add.f64         %fd21, %fd20, %fd19;
st.volatile.shared.f64  [%r4], %fd21;
setp.lt.u32     %p11, %r11, 32;
@%p11 bra       $L__BB0_21;
ld.volatile.shared.f64  %fd22, [%r4];
ld.volatile.shared.f64  %fd23, [%r4+128];
add.f64         %fd24, %fd23, %fd22;
st.volatile.shared.f64  [%r4], %fd24;
setp.lt.u32     %p12, %r11, 16;
@%p12 bra       $L__BB0_23;
ld.volatile.shared.f64  %fd25, [%r4];
ld.volatile.shared.f64  %fd26, [%r4+64];
add.f64         %fd27, %fd26, %fd25;
st.volatile.shared.f64  [%r4], %fd27;
setp.lt.u32     %p13, %r11, 8;
@%p13 bra       $L__BB0_25;
ld.volatile.shared.f64  %fd28, [%r4];
ld.volatile.shared.f64  %fd29, [%r4+32];
add.f64         %fd30, %fd29, %fd28;
st.volatile.shared.f64  [%r4], %fd30;
setp.lt.u32     %p14, %r11, 4;
@%p14 bra       $L__BB0_27;
ld.volatile.shared.f64  %fd31, [%r4];
ld.volatile.shared.f64  %fd32, [%r4+16];
add.f64         %fd33, %fd32, %fd31;
st.volatile.shared.f64  [%r4], %fd33;
setp.lt.u32     %p15, %r11, 2;
@%p15 bra       $L__BB0_29;
ld.volatile.shared.f64  %fd34, [%r4];
ld.volatile.shared.f64  %fd35, [%r4+8];
add.f64         %fd36, %fd35, %fd34;
st.volatile.shared.f64  [%r4], %fd36;

```

Listing 15: `warpReduce2()` assembly code

```
ld.volatile.shared.f64  %fd10, [%r4];
ld.volatile.shared.f64  %fd11, [%r4+8];
add.f64         %fd12, %fd11, %fd10;
st.volatile.shared.f64  [%r4], %fd12;
```

Listing 16: `warpReduce1<2>()` assembly code. Almost all of the logic of listing 12 is precomputed by the compiler.

Since the Lanczos algorithm is comprised of many calls to `spMV(...)`, `dot_prod(...)`, `norm(...)`, which all make use of reductions, the warpReduce operation is a significant computational kernel for our algorithm. This necessitates a high level of optimisation. Extra work with templated parameters yields high performance benefits, at a small cost of having to deal with long lists of explicit instantiations of templates—marginal code bloat.

7 Scaling

The CUDA Lanczos approximation behaves and performs very well when the graph in question fits in the memory of a single GPU. However, graphs can grow almost arbitrarily large, so it is necessary to understand how the method will behave on multiple GPUs. Please see `parallel-two-cards/` for implementation.

For a large, sparse graph we can assume that $|E| > n$, and that the CSR representation of A does not fit in the global memory of a single card. We will assume that a vector of size n *does* fit in the global memory of the GPU. Assume that the CSR representation of A as well as all the necessary vectors can be split between the global memory of two cards.

Since it is desirable to minimize transfers between two GPUs, the only operation that will be executed by *both* cards will be the most time-complex operation in each iteration of the Lanczos method: SPMV.

See Figure 23 for an illustration. In this implementation each GPU holds a partition of A, where GPU1 holds A_1, or the first $n/2$ rows of A, and GPU2 holds A_2, the last $n/2$ rows of A. In steps:

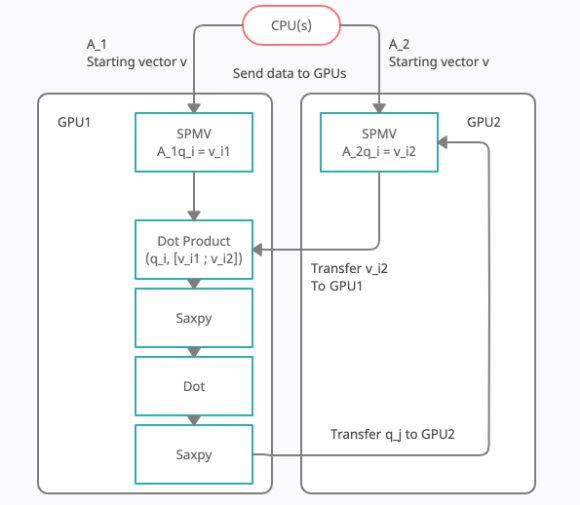


Figure 23: Lanczos decomposition on two GPUs. Author's creation using `creately.com`

1. Each GPU performs its own local SPMV: $A_1 q_i = v_{i1}$ and $A_2 q_i = v_{i2}$.
2. GPU2 sends v_{i2} to GPU1, and GPU1 concatenates v_{i1} and v_{i2} into v_i.
3. GPU1 completes the rest of the cycle on card.
4. GPU1 sends the new basis vector q_{i+1} to GPU2 in order to perform the SPMV operation again.
5. Repeat if $i < r$.

Each iteration needs two memory transfers between cards. The first (step 2) sends a vector of size $n/2$ from GPU2 to GPU1, and the second (step 4) sends a vector of size n from GPU1 to GPU2.

In `parallel-two-cards` run `make lanczos_test` to see sample performance of the two card Lanczos process for small matrices.

Focussing on "Stream 14" row in figure 24, which is the stream responsible for all compute on GPU1, we can see that most of the time is spent waiting for data, rather than computing. There is no way around this data wait, if a two card implementation is indeed desirable, since all Lanczos operations are sequential. For comparison see figure 25 in section 10.

Matrix	NotreDame yeast	California	Small Barabasi
Speedup vs serial	0.055	0.32	0.505

Table 4: Speedups for 2xGPU Lanczos method vs serial

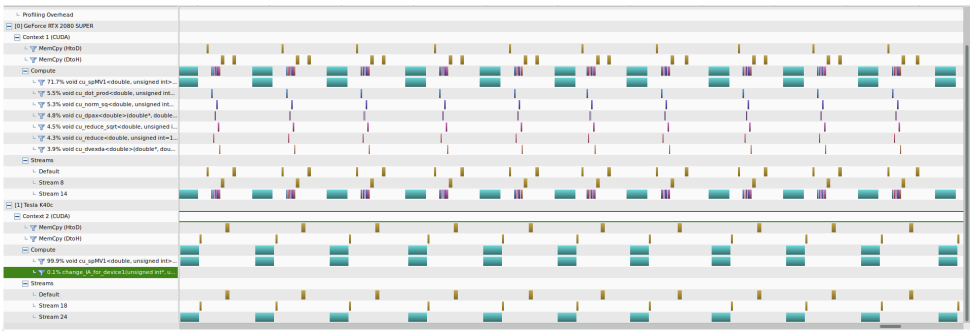


Figure 24: The two GPU Lanczos algorithm. Compare with the single card implementation seen in Figure 25.

Conclusion While these tests are admittedly cursory, from examining the speedups in table 4 it is clear that the two card method is currently not feasible, at least on this particular machine (see section B). The poor performance is caused by a two main factors.

- The need to compile *all* code to a lower standard (since the secondary GPU has compute capability much lower than the primary GPU, and all device code must be of the same standard). This means that the GeForce will be running its portion of the code slower than its speed if it were running alone.

- The traffic between cards. Data transfer at every iteration is a big hindrance to performance. Especially since in the machine that is being used for this project, GPU-to-GPU communication must travel through the CPU, meaning each transfer is in fact two trips on the PCI express bus. This makes the active transfer time of peer-to-peer communication twice as long as that for normal memory transfers from device to host or vice versa. Moreover, latency can not be reduced in any meaningful way by using streams, since all memcpys contain data that the receiving card needs in order to begin the next step in the algorithm.

The author relishes the opportunity to perform these tests on a different hardware setup, especially one with Remote Direct Memory Access (RDMA) connecting two or more GPUs. See [19] for a treatment. If the gaps in figure 24 can be reduced significantly, then maybe this method will become scalable on multiple GPUs.

Perhaps in the future multi-card implementations will become more feasible. But for the moment (or at least with limited hardware) the programmer must be content to perform the Lanczos approximation on a single GPU if performance is desired.

Graph sampling [20] is another means of dealing with graphs too large for the global memory on a single card. This method allows approximations to matrix functions to be computed on

subgraphs rather than on an entire graph. This will be an area of further research for the author.

8 Final Implementation

This section was re-run after the experiment in section 9. Therefore all runs in this section use the `free_mem()` function detailed therein.

For memory considerations, the memory light `cu_SPMV1` was preferred over the load balancing `cu_SPMV2` in the final implementation of the Lanczos method.

Please see `main.cu` in `parallel-final` for final implementation of the Lanczos algorithm. The implementation uses a CUDA Lanczos method, as well as parallel BLAS routines for multOut. Use the command `make run` to run code.

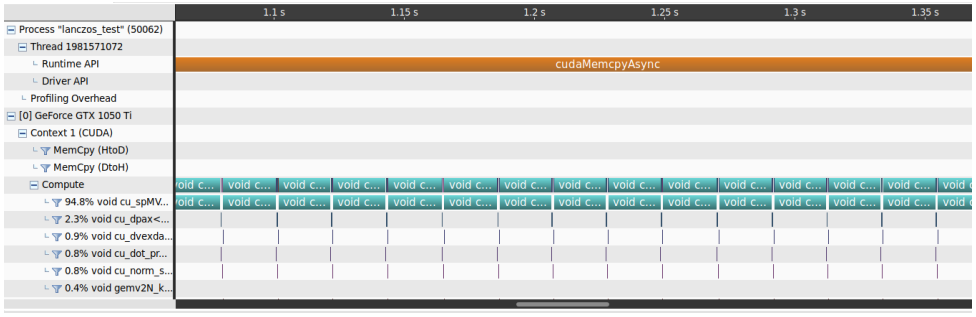


Figure 25: The Lanczos iteration on one GPU. The method achieves far higher occupancy than the two-card method seen in figure 24.

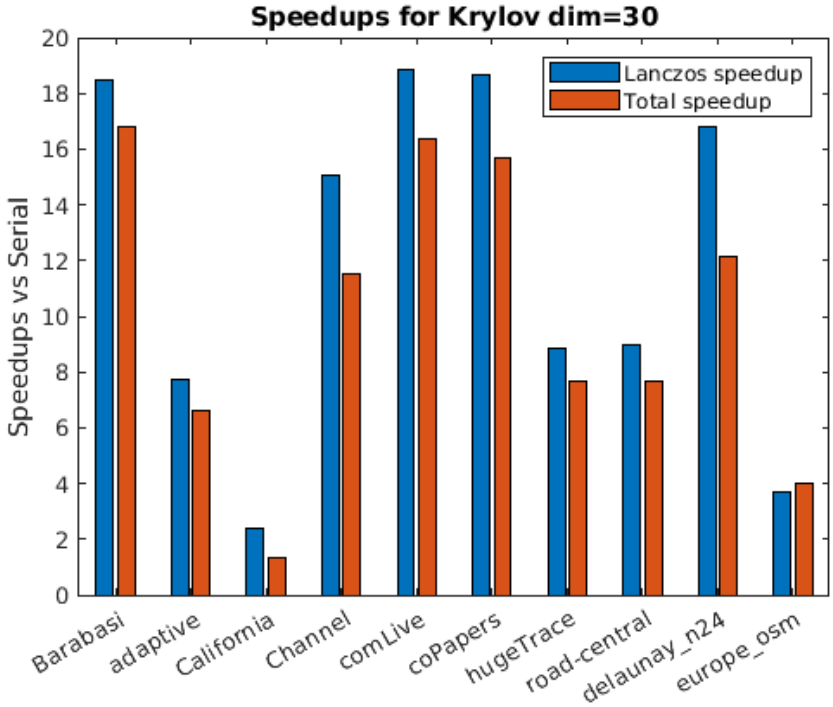


Figure 26: Speedups vs serial for Krylov dim=30

The code gains good speedups for all medium to large size matrices. Note that for California

(see appendix A), which has $n = 9664$, the speedups are not as dramatic as for larger matrices.

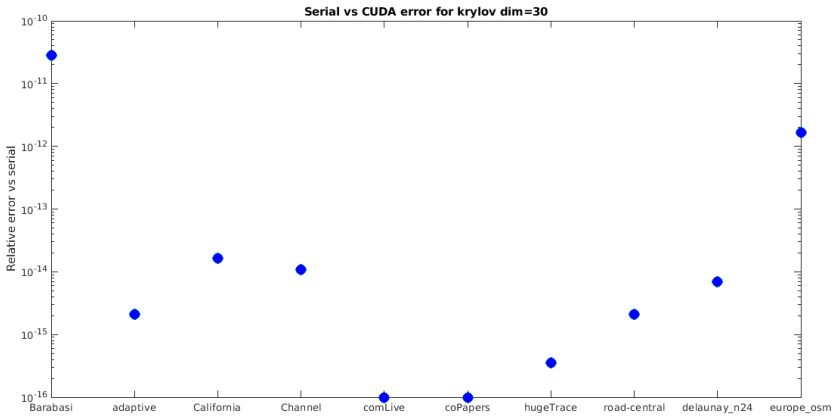


Figure 27: Relative error terms vs serial computation for Krylov dim=30. To calculate the error, the norm of the difference between both vectors was taken, and divided by the norm of the serially-generated answer.

See in figure 27 that the Barabasi matrix (see appendix A), a matrix constructed to be imbalanced, gives the highest relative error vs serial computation. It is worth bearing in mind, therefore, that results may be slightly inaccurate dependent on how poorly conditioned A is. Note that Barabasi is an *extreme* example and only one of the other matrices (Europe OSM) had relative errors above around 1e-14.

See `parallel-final/final_output.txt` for a full output from final runs of code.

9 Breaking the Code

There are two ways of dealing with Q on the GPU:

1. Compute and store the entirety of Q on the GPU, and then send it back in a single transaction to the CPU for multOut. This requires `sizeof(T)*n*krylov_dim` bytes to store `Q_d`. This places a limit on the size of the matrix we can compute on, as well as the Krylov dimension for large matrices.

2. Send each column of Q back to host memory as it is formed on the GPU. This requires `sizeof(T)*n*2` bytes in global memory to store the two most recent columns of Q. The `memCpy` operation can be done asynchronously, making the operation as fast as approach (1), as long as the memcpy has finished before the column being transferred is needed to be changed out by the Lanczos algorithm. There will be a device to host memcpy at each cycle of size `sizeof(T)*n`.

In order to try and break the code with large data we will use method number 2, so we can accommodate larger Q matrices than would be possible with method 1. We will also revert to using the memory light `cu_SPMV1`. Since `blockrows_d`, needed by `cu_SPMV2`, adds another `sizeof(T)*n` to the memory requirement of the method.

9.1 Breaking

With that in mind, let's see how the method behaves with the large matrix "Europe OSM" see appendix A. Clearly, approach (1) for storing Q on card is not feasible since this would greatly limit the Krylov dimension. Therefore approach 2 is taken. Note the speedups.

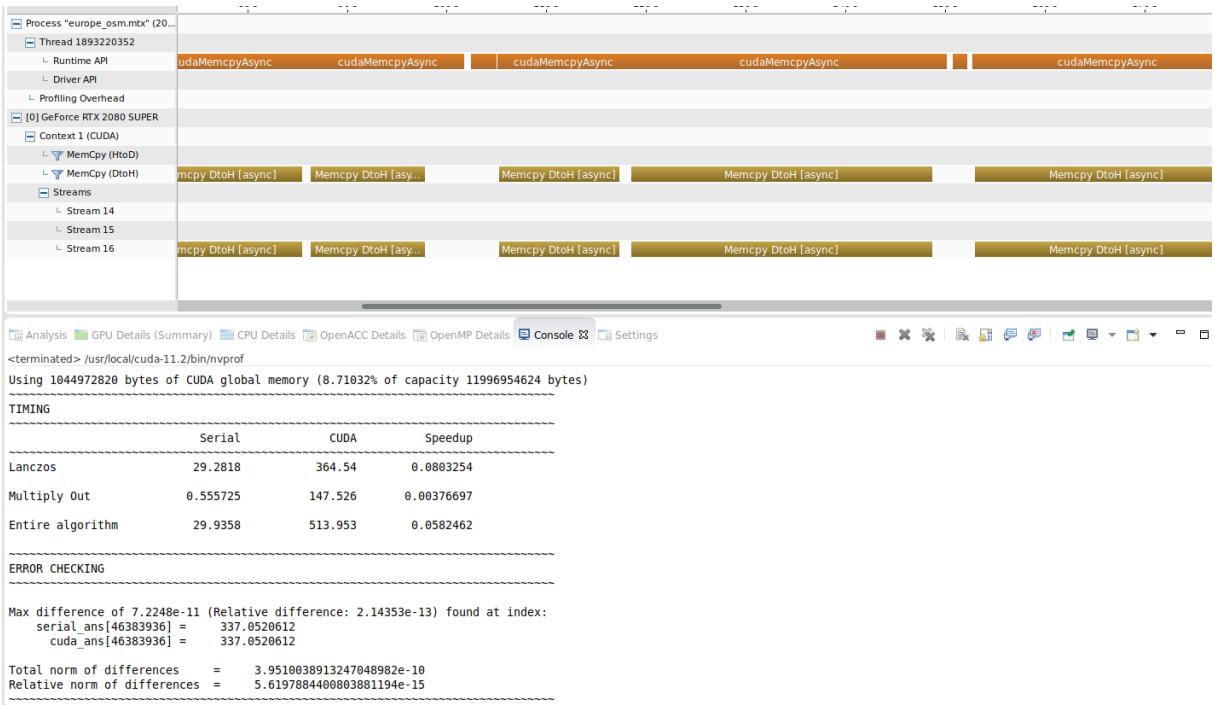


Figure 28: Output and profile of Lanczos running for Europe OSM

See from the output (bottom left of Figure 28) that the CUDA Lanczos method is more than 10x slower than serial! A terrible result of course. Examining the profile (top of Figure 28) we see that the majority of the time in the Lanczos method is taken up with Device to Host Memcpys. Very little of the timeline is, in fact, *not* a memcpy.

For this particular example (Europe OSM) $n = 50,912,018$, meaning that at each iteration there is a Device to Host memcpy of `sizeof(T)*50912018` bytes. Which is 407,296,144 bytes when we are dealing with `doubles`. The bandwidth of the PCI-Express 4.0 is stated as 32GB/s, meaning that this operation should execute in less than 1/64th of a second, not accounting for latency. This 1/64th of a second is unlikely to be the only cause of hamstrung performance, since in figure 28 each memcpy lasts for more than a second. It seems that the problem is rather the loading of the memory from device into swap space on the Host (see figure 29 to see the swapspace filling up). This slow loading of Q into host memory only occurs for the CUDA implementation and not for the serial version, for the arbitrary reason that the serial code is run first in this particular instance, and the serial Lanczos's memory is not freed before the CUDA version begins executing.

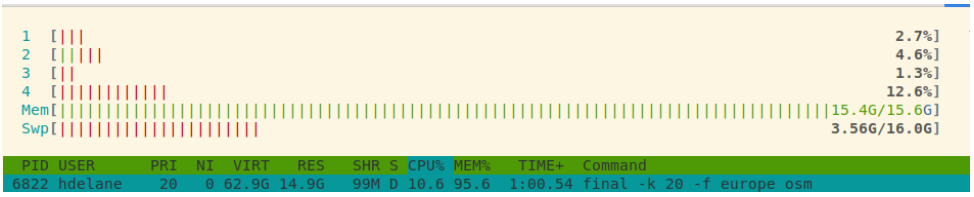


Figure 29: Swap space filling up as memory is passed from Device to Host. RAM is full as can be seen in "Mem" row.

9.2 Fixing

Adding a function `free_mem()`, allows us to free up the memory used by `Q`, and other arrays contained within the `lanczosDecomp` object, allowing us to get back some memory before we want the Lanczos object to go out scope (we want the Lanczos object to stay in scope just so we can compare the answer vectors). Now freeing up some memory and performing the same operation:

```
[hdelane@cuda01 parallel-final]$ ./final -f europe_osm -k 20
Going to open file: ../data/europe_osm/europe_osm.mtx
Time elapsed to build random adjacency matrix with n = 50912018 edges = 54054660:
  40.6953 seconds

Running Lanczos algorithm for krylov_dim 20

Using 1044972820 bytes of CUDA global memory (8.71032% of capacity 11996954624 bytes)
--------------------------------------------------------------------------------
TIMING
--------------------------------------------------------------------------------
                            Serial              CUDA          Speedup
--------------------------------------------------------------------------------
Lanczos                     29.3569           4.14441           7.0835

Multiply Out               0.546276          0.765599          0.713528

Entire algorithm            29.9034           4.91012           6.09015

--------------------------------------------------------------------------------
ERROR CHECKING
--------------------------------------------------------------------------------

Max difference of 7.2248e-11 (Relative difference: 2.14353e-13) found at index:
    serial_ans[46383936] =      337.0520612
      cuda_ans[46383936] =      337.0520612

Total norm of differences     =      3.9510038913247048982e-10
Relative norm of differences  =      5.6197884400803881194e-15
```

Listing 17: Output after freeing memory from the first Lanczos iteration.

The desired result!

Freeing up memory after each decomposition allows the method to maintain decent performance for larger graphs, however this experiment illustrates an important principle: *Device to Host memcpys that overflow RAM will be extremely slow, and ought to be avoided at all costs.* Although harddisk space ought to be fast enough to accommodate data transfers, we may posit that in this case lots of memory "juggling" has had to take place in order to get the memory from global memory to host swapspace, leading to very poor performance. Host code may have more streamlined avenues in order to move memory into swapspace, which doesn't damage performance as much as it initially did for the CUDA implementation. However, any use of swapspace will damage performance markedly.

9.3 Memory Light Implementation

9.3.1 `floats`

For large matrices, another way of extending the range of the method is to use single precision floats instead of doubles, which have been used until this point. First, to compare the relative error between a single and double precision approximation.

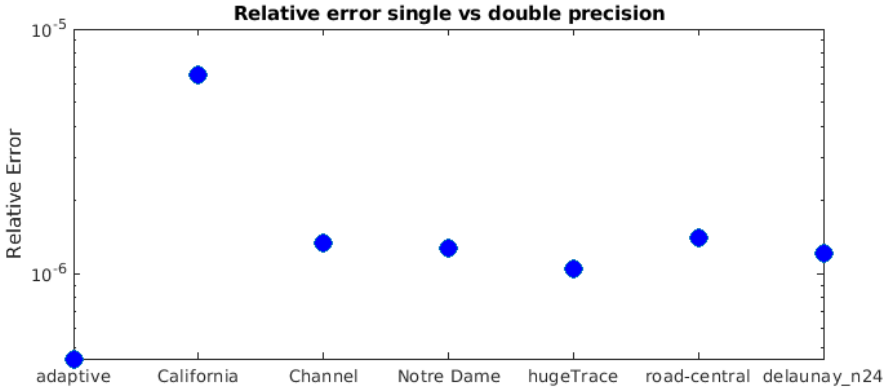


Figure 30: Relative error terms of single vs double precision Lanczos approximations.

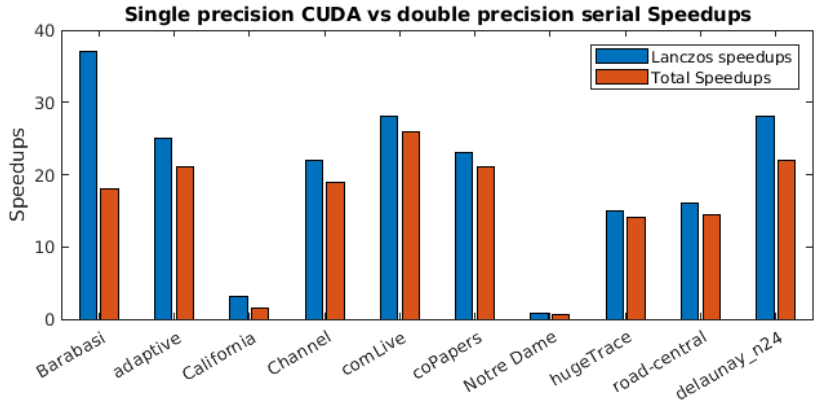


Figure 31: Single precision CUDA vs double precision serial speedups.

See `output/single_double.txt` for full output. From figure 30 it is clear that error terms between single and double precision approximations are about as good as can be expected. However, in `output/single_double.txt`, we also see that single precision approximations for matrices: Barabasi, CoPapers, comLive, had `nan` entries, due to floating point overflow. This is a danger when using single precision, especially for denser matrices for which $e^A x$ has large entries. Precautions must be taken when using single precision to check for floating point overflow.

Using single precision means that the range of computation for the Lanczos approximation is effectively doubled, meaning larger matrices can be computed on with better performance but lesser accuracy than the double precision counterpart. We can also see in figure 31 that using single precision gives very good speedups over double precision in serial.

Matrix	norm(ans35-ans30)/norm(ans35)
Adaptive	0
Barabasi (bn1000000e9999944)	1.822e-7
California	1.82e-14
Channel500	0
coPapersDBLP	3.2e-7
Delaunay n24	0
Europe OSM	0
Hugetrace	0
Road Central	0

Table 5: Approximations to $e^A x$ compared for Krylov dimensions 30, 35. Norms computed with MATLAB.

10 Results

10.1 Convergence

In `data/*/` see files `ans30.txt`. In this case the 30 refers to the Krylov dimension of the Lanczos decomposition.

Since Q is a dense matrix of dimension ($n \times$ Krylov dim), the dimension of the Krylov subspace will be constrained for large matrices, since we do not want Q to get too large for memory (which may cause a segmentation fault, or worse—computing in swapspace!). Therefore we must check how the Lanczos approximation converges for large matrices; whether it suffices to take a small Krylov dimension for large matrices. While in section 4.3 we were able to check the approximation's convergence to an analytic answer, here we must be content to allow the approximation to converge to *something*, and trust that it is a highly accurate approximation of $e^A x$.

In Table 5 we see that the convergence of the algorithm is, on the whole, very good for Krylov dim = 30 or 35. Note as well that the dimension of the Krylov subspace necessary for convergence does not seem to depend on the dimension of the matrices. California, despite being the smallest matrix, needs one of the largest rank Krylov subspaces, despite its dimension being several orders of magnitude smaller than the other matrices. Barabasi, the self generated imbalanced matrix, is one of the last to converge. Perhaps imbalanced matrices require greater rank Krylov subspaces in order to get good approximations. More research is needed on the author's part.

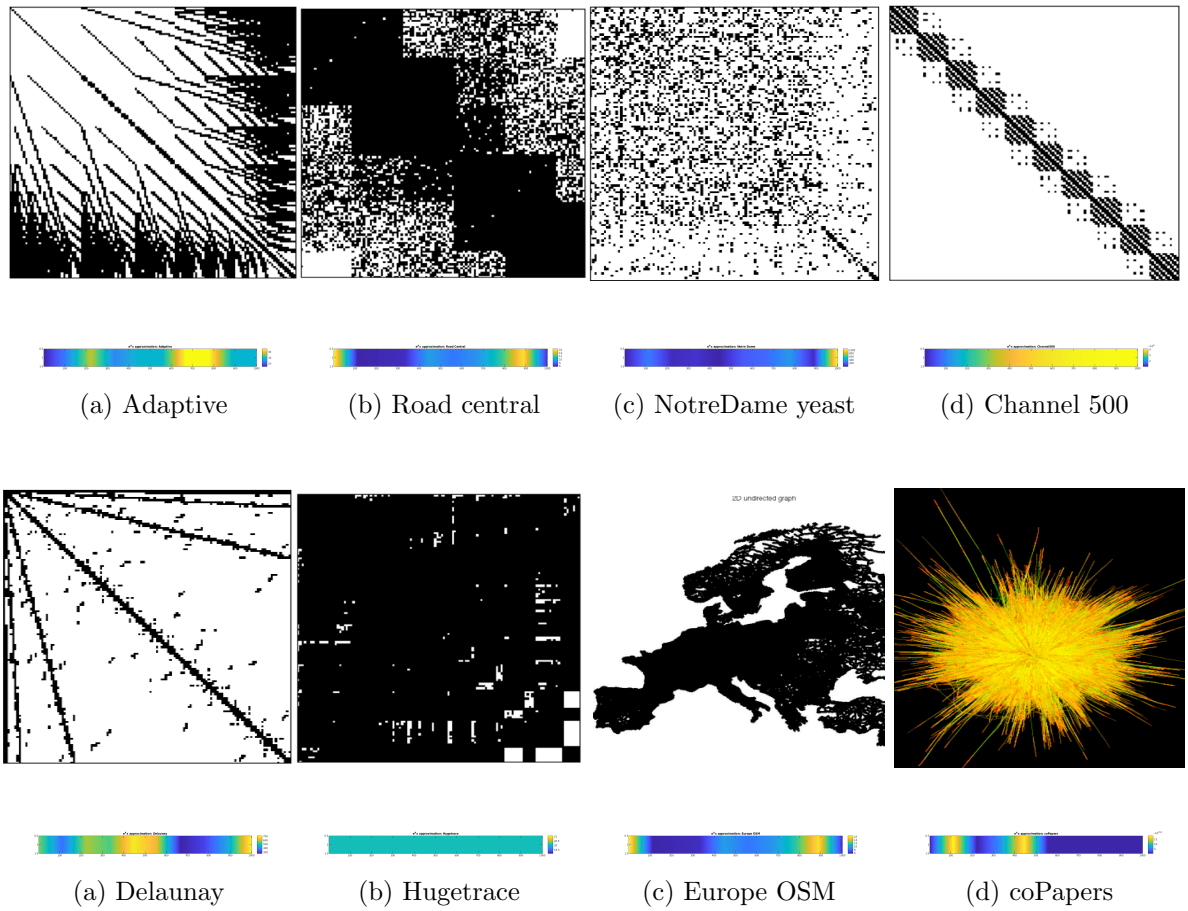
This convergence demonstrates that we can be justified in using low rank approximations for $e^A x$. However, it also means that it is not entirely clear when larger rank subspaces are necessary. Note that for the seemingly incongruous coPapers matrix:

```
norm(ans150-ans100)/norm(ans(150) = 9e-14
```

Note that coPapers is the densest matrices that this project has dealt with, with average

$nnz/row = 56.414$. This results in many entries per row of the abstract matrix e^A, which in turn gives $e^A x$ a high value (around 1e+147) in this case. The denseness of the matrix may be the biggest contributing factor to the slow convergence, since perhaps $e^A x$ is not as easily represented by a handful of basis vectors. More research is needed.

10.2 Output



(a) Adaptive (b) Road central (c) NotreDame yeast (d) Channel 500

(a) Delaunay (b) Hugetrace (c) Europe OSM (d) coPapers

Figure 35: Colourbars are heatmaps of the Lanczos Approximations to $e^A x$. Note that the vector $e^A x$ has been interpolated for visualization purposes. The true approximation is not continuously coloured.

Please peruse the answer vectors in `data/`. The outputted approximations are far more fine-grained than can be reperesented on paper.

Examining figure 35b, as well as `data/hugetrace-00020/ans50.txt`, it seems that the Lanczos approximation to $e^A x$ has output a vector where *nearly* all of the entries are 20.0855. Note that some values are indeed different, but only 86,281 out of the total 16,002,413. Here the centrality measure of $e^A x$ was of limited use.

11 Conclusion

The Lanczos method is an incredibly powerful iteration for the analysis of undirected graphs. The accuracy of the method as well as its fast convergence makes it one of the most scalable algorithms in computing graph centrality measures. Nevertheless, like all algorithms and methods, it has a range in which the method is effective, and a range in which it is not effective. Moreover, there are inputs (like Hugetrace for instance) where the measure $e^A x$ informs us very little about the structure of the graph. Judicious and observant application of the Lanczos

method is advised; when dealing with large, irregular and unique matrices there is seldom a one-size-fits-all solution.

Memory The Lanczos method is incredibly effective for matrices where `Q` will fit in RAM. Our calculations suggest that a Krylov dimension of 30 may be good enough to give convergence of $e^A x$ for *the average matrix*, although no assumptions may be made as to whether the matrix one is computing on is indeed an *average* matrix. It is a significant challenge for those who use the Lanczos approximation to clarify whether the Krylov approximation, small enough to fit in RAM, is indeed a good enough approximation to $e^A x$. Convergence must be analysed, which may be done by performing a Lanczos decomposition of rank r, then iteratively performing multOut for the first $k < r$ columns of Q and α, β values and seeing how the approximation changes.

The programmer must be explicitly aware of the limits of their own hardware when computing the Lanczos approximation, most notably the size of GPU global memory and RAM. A naive programmer may baffle at the incredible slowdown as the program enters swapspace, so a keen awareness is required.

While additional RAM may extend the applicable range of the Lanczos method, also valuable is the ability to swap to single implementation, which similarly allows bigger matrices to be computed on.

Parallelization Parallelization is rewarded highly for the Lanczos approximation; good speedups were achieved in CUDA and the most simple implementation ended up being entirely appropriate to the task at hand. However the speed and efficacy of the Lanczos approximation begs the question *"Why parallelize a method that only iterates 30 times?"*

One can imagine a few scenarios:

- A method that needs to analyze a multitude of small to medium sized graphs would benefit greatly from a parallel method. A 15x speedup on 30 iterations becomes significant when said 30 iteration need to be computed on 10^9 graphs.
- Graph sampling. While the author is not an expert in the field, it seems plausible that in order to get centrality measures for a very large graph, a method could rapidly draw medium sized subgraphs from the graph and compute local Lanczos approximations to $e^A x$, then merge the results in some fashion or another. This will be an area of research for the author going forward.

There are no doubt other methods in which an extremely fast approximation of $e^A x$ is required.

As seen in section 7, parallelization that requires communication between nodes or separate GPUs is currently very difficult to optimise for using the Lanczos method. This is paradoxically because the algorithm is so efficient (running in $O(k|E|)$ time), that a matrix massive enough to necessitate partitioning among nodes, would require massive data-transfers at each iteration, which hamstrings perfomance.

Final Thoughts The Lanczos iteration has a well defined *sweet spot*. There is a size of matrix for which the Lanczos iteration achieves amazing results efficiently and easily. This range can be extended by upgraded hardware and downcasting to `float`s. However, these methods merely push the frontier back by a factor of two or so, and eventually limits will be reached.

Instead of seeking to extend the limits by increments, more pressing is the question: how does one take a large matrix for which the Lanczos method is unsuitable, and represent it as a smaller matrix for which the Lanczos approximation is speedy and efficient? Sampling from large graphs is completely necessary, and requires not only Krylov methods but also traversal methods as outlined in section 1.3.2. The question of how to combine these disparate approaches has ignited the author's curiosity, and will be an area of continuing research.

Appendix

A Data

All graphs are undirected graphs stored in `.mtx` format. Most are taken from the SuiteSparse Matrix Collection `https://sparse.tamu.edu`.

Graph name	Rows	Columns	Nonzeros	Avg nonzeros/row	Source
Barabasi (bn...)	1,000,000	1,000,000	9,999,944	9.99	Self-generated
California	9,664	9,664	16,150	1.6711	`sparse.tamu.edu`
Channel500	4,802,000	4,802,000	85,362,744	17.7764	`sparse.tamu.edu`
CoPapers	540,486	540,486	30,491,458	56.414	`sparse.tamu.edu`
ComLive Journal	3,997,962	3,997,962	69,362,378	17.349	`sparse.tamu.edu`
Delaunay_n24	16,777,216	16,777,216	50,331,601	2.999	`sparse.tamu.edu`
Europe_osm	50,912,018	50,912,018	54,054,660	1.071	`sparse.tamu.edu`
kmer_U1a	67,716,231	67,716,231	69,389,281	1.024	`sparse.tamu.edu`
Hugetrace	24,575	24,575	11,111,110	3.164	`sparse.tamu.edu`
NotreDame_yeast	2,114	2,114	2,277	1.077	`sparse.tamu.edu`
Road Central	14,081,816	14,081,816	33,866,826	2.405	`sparse.tamu.edu`

Barabási-Albert Networks are generated by the function `barabasi()`, which can be found in `lib/make_graph.cc`. Barabási graphs are constructed based on a preferential attachment principle. Nodes are joined to the graph one-by-one, where each node has a fixed number of edges that it must attach to the existing graph. Edges of the new node choose which nodes to attach their edges to by means of a probability distribution, where:

$$p(\text{New node makes edge with existing node } i) = \frac{\text{degree}(i)}{2(\#\text{ edges in graph})}$$

This leads to highly imbalanced graphs; as the graph grows the well connected nodes become better connected, and the poorly connected nodes have little likelihood of becoming better connected. Extreme Barabási-Albert graphs will be some of the most poorly-behaved graphs we can expect to deal with. Therefore it is good fodder for the Lanczos algorithm, and especially the SPMV kernels to see which kernels can balance load most efficiently.

B Hardware

The computer used for all calculations has 16G of RAM.

B.1 GPUs

	Primary GPU	Secondary GPU
Model	Nvidia GeForce RTX 2080 Super	Nvidia Tesla K40c
Cores/CUDA cores	3072	2880
Global Memory	8GB	12GB
Compute Capability	7.5	3.5
PCI Express	4.0	4.0

All single card code was run on the GeForce 2080. The K40 was only utilized in addition to the GeForce in section 7

B.2 CPU

The CPU used is an Intel Core i5-3570K with 4 physical cores.

References

[1] Benzi, M. (2020). *Matrix Functions in Network Analysis*, Scuola Normale Superiore di Pisa.

[2] Harris, M. (2010). *Optimising Parallel Reduction in CUDA*, `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`

[3] Brandes, U. (2001). *A Faster Algorithm for Betweenness Centrality*, Journal of Mathematical Sociology 25, pp. 163-177.

[4] Trefethen, L. N. and Bau D. III. (1997). *Numerical Linear Algebra*, The Society for Industrial and Applied Mathematics.

[5] Moler, C. and Van Loan, C. (2003). *Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later*, SIAM Review, Vol 45, No. 1.

[6] Quinlan, R. *Diagonalizability of Symmetric Matrices*, `https://maths.nuigalway.ie/~rquinlan/linearalgebra/section2-2.pdf`

[7] Saxena, A., Gera, R. and Iyengar, S. R. S (2017), *A Faster Method to Estimate Closeness Centrality Ranking*, `https://arXiv.org/pdf/1706.02083v1.pdf`

[8] Lumsdaine, S., Gregor, D., Hendrickson, B. and Berry, J. (2007), *Challenges in Parallel Graph Processing*, Parallel Processing Letters, 1 March 2007, World Scientific.

[9] Oracle Inc., *Pagerank Algorithm Reference*, `https://docs.oracle.com/cd/E56133_01/latest/reference/analytics/algorithms/pagerank.html`

[10] Strang, G. (2006), *Krylov Methods and Conjugate Gradients*, `https://ocw.mit.edu/courses/mathematics/18-086-mathematical-methods-for-engineers-ii-spring-2006/readings/am64.pdf`

[11] Saad, Y (2003), *Iterative Methods for Sparse Linear Systems, Second Edition*, The Society for Industrial and Applied Mathematics.

[12] Benner, P. and Fassbender, H (2000), *An Implicitly Restarted Symplectic Lanczos Method for the Symplectic Eigenvalue Problem*, SIAM Journal on Matrix Analysis and Applications 22(3) pp. 682-713.

[13] Van Loan, C. F., (1977), *The Sensitivity of the Matrix Exponential*, SIAM Journal on Numerical Analysis, 14 pp. 971-981.

[14] Musco, Christoper; Musco, Cameron; and Sidford, A (2017), *Stability of the Lanczos Method for Matrix Function Approximation* `https://arxiv.org/pdf/1708.07788.pdf`

[15] Wu, K. and Simon, H. (1997), *A Parallel Lanczos Method for Symmetric Eigenvalue Problems*, Lawrence Berkeley National Laboratory, NERSC, Berkeley.

[16] Gao, J., Qi, P. and He, G (2016), *Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU*, Hindawi Mathematical Problems in Engineering, article 4596943.

[17] Zhang, L., Wahib, M., Zhang, H., and Matsuoka, S. (2020) *A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs*, `https://arxiv.org/pdf/2004.05371.pdf`

[18] Luitjens, J. (2014), *Faster Parallel Reductions on Kepler*, `https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/`

[19] Nvidia Developer, *GPUDirect RDMA*, `https://developer.nvidia.com/gpudirect`

[20] Leskovec, J. and Faloutsos, C. (2006) *Sampling from Large Graphs*. `https://cs.stanford.edu/~jure/pubs/sampling-kdd06.pdf`