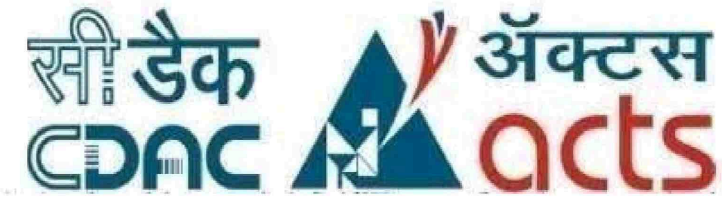
Project Report

Parallelizing Google's PageRank algorithm in C++ with CUDA framework on GPU.



Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in High Performance Computing Application Programming

(C-DAC, ACTS (Pune))

Guided By:

Mr. Saurabh Deshkar

Submitted By:

Akshay Ghadge	(230940141005)
Kartik Narayane	(230940141009)
Ketan More	(230940141010)
Rohan Wagh	(230340141022)
Siddhant Gedam	(230940141024)
Vikas Gatkhane	(230940141027)

Centre for Development of Advanced Computing
(C-DAC), ACTS (Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr. Saurabh Deshkar** C-DAC, Pune for his constant guidance and helpful suggestion for preparing this project **Parallelizing Google's PageRank algorithm in C++ with CUDA framework on GPU.**
We express our deep gratitude towards his for inspiration, personal involvement, constructive criticism that he provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Swati Salukhe (Course Coordinator, PG-HPCAP)** for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Akshay Ghadge	(230940141005)
Kartik Narayane	(230940141009)
Ketan More	(230940141010)
Rohan Wagh	(230940141022)
Siddhant Gedam	(230940141024)
Vikas Gathkhane	(230940141027)

ABSTRACT

On a daily basis, we rely on the Internet for accessing information. Utilizing search engines, we navigate through the vast expanse of online content to locate valuable insights. This functionality is facilitated by algorithms like PageRank, which evaluate the significance of web pages. PageRank algorithm systematically assesses web pages, assigning them ranks based on predefined criteria. Search engines then present search results based on these rankings, prioritizing pages with higher popularity scores. Traditional methods employ multi-CPU architectures, yet these come with drawbacks such as communication overhead and limited CPU processing power compared to GPUs. Therefore, optimizing the PageRank algorithm for parallel GPU-CPU environments proves advantageous, enhancing energy efficiency and reducing computation time for evaluating PageRank positions across extensive web graphs.

Table of Contents

Chapter 1
Introduction

1.1 Introduction

In today's digital age, the accessibility and relevance of online information play a crucial role in shaping our daily lives. Search engines serve as the gateway to this vast sea of data, allowing users to navigate through a multitude of web pages to find the information they seek. Among the myriad of algorithms employed by search engines, Google's PageRank algorithm stands out as a pioneering approach to ranking web pages based on their importance and relevance.

The PageRank algorithm, developed by Larry Page and Sergey Brin at Google, revolutionized the way search engines prioritize and present search results. By analyzing the link structure of the web, PageRank assigns each web page a numerical value, indicating its importance in the network of interconnected pages. Pages with higher PageRank scores are deemed more authoritative and are thus displayed more prominently in search results.

While PageRank has proven to be a powerful tool for organizing and ranking web content, its computation can be computationally intensive, especially when dealing with large-scale web graphs. Traditional implementations of the PageRank algorithm often face performance bottlenecks, particularly in terms of processing speed and scalability.

To address these challenges, researchers have explored various optimization techniques, including parallel computing architectures like CUDA (Compute Unified Device Architecture). CUDA, developed by NVIDIA, enables the utilization of the massive parallel processing power of Graphics Processing Units (GPUs) to accelerate compute-intensive tasks.

In this report, we delve into the optimization of Google's PageRank algorithm using CUDA, aiming to leverage the parallel computing capabilities of GPUs to enhance performance and scalability. By harnessing the computational prowess of CUDA-enabled GPUs, we seek to expedite the computation of PageRank scores for large web graphs, thereby improving the efficiency and effectiveness of search engine operations. Through a comprehensive analysis and implementation of CUDA-accelerated PageRank algorithm, we aim to contribute to the advancement of search engine technology and information retrieval systems.

Parallelizing Google's PageRank algorithm in C++ with CUDA framework on GPU.

---

1. **Understanding PageRank Algorithm** : Before diving into optimization techniques, it's essential to have a solid understanding of how the PageRank algorithm works. This includes comprehending the underlying principles of link analysis, random walks, and the significance of inbound links in determining a page's importance.

2. **Challenges with Traditional Implementations** : Traditional implementations of the PageRank algorithm often face challenges related to scalability and computational efficiency. As web graphs grow larger and more complex, the computational demands required to calculate PageRank scores increase exponentially. This leads to longer processing times and resource constraints, hindering real-time application of the algorithm.

3. **Introduction to CUDA** : CUDA is a parallel computing platform and programming model developed by NVIDIA for GPU-accelerated computing. It allows developers to harness the massive parallel processing power of GPUs to accelerate a wide range of computational tasks. Understanding the fundamentals of CUDA programming, including kernel execution, memory management, and thread synchronization, is crucial for optimizing algorithms like PageRank.

4. **Parallelization Strategies** : The key to optimizing PageRank using CUDA lies in effectively parallelizing the algorithm to leverage the parallel processing capabilities of GPUs. This involves partitioning the computation into independent tasks that can be executed concurrently on multiple GPU cores. Strategies such as vertex-centric parallelism, edge-centric parallelism, and hybrid approaches can be explored to maximize parallel efficiency and minimize communication overhead.

5. **Memory Optimization** : Memory access patterns and data movement can significantly impact the performance of CUDA-accelerated algorithms. Optimizing memory access, utilizing shared memory for inter-thread communication, and minimizing global memory transactions are essential for maximizing memory bandwidth utilization and reducing latency.

1.2 Objective:

The objective of the PageRank algorithm is to rank web pages based on their importance and relevance. It was originally developed by Larry Page and Sergey Brin, the co-founders of Google, and is a key component of Google's search engine algorithm.

The PageRank algorithm works by analyzing the link structure of the web. It assigns a numerical weight to each page, with the purpose of measuring its relative importance within the network of web pages. Pages with higher PageRank scores are considered more important and are likely to appear higher in search engine results.

When implementing the PageRank algorithm with a CUDA program, the objective is to leverage the parallel processing capabilities of CUDA-enabled GPUs to accelerate the computation of PageRank scores for a large number of web pages.

By utilizing the parallel processing power of GPUs, the algorithm can be executed more efficiently, leading to faster computation times and improved scalability for handling large-scale web graphs.

The main objective of using CUDA for the PageRank algorithm is to achieve significant performance improvements by harnessing the parallel processing capabilities of GPUs, thereby enabling the efficient computation of PageRank scores for large web graphs.

Chapter 2

LITERATURE REVIEW

The optimization of Google's PageRank algorithm through parallelization using CUDA has been a subject of significant research interest in recent years. Several studies have explored various parallelization strategies, performance optimizations, and real-world applications of accelerating PageRank computation using GPUs. This literature review provides an overview of key contributions in this field

In conclusion, the literature on parallelizing Google's PageRank algorithm using CUDA showcases significant advancements in leveraging GPU acceleration to enhance the performance and scalability of PageRank computation. By exploring parallelization strategies, memory optimization techniques, and real-world applications, researchers continue to push the boundaries of GPU-accelerated PageRank optimization, paving the way for more efficient and scalable information retrieval systems.

GPU Acceleration for PageRank Computation : Researchers have explored the potential of GPU acceleration for speeding up PageRank computation. GPGPU (General-Purpose computing on Graphics Processing Units) frameworks like CUDA have been utilized to parallelize the iterative computation involved in PageRank. Studies such as [1] have demonstrated substantial speedups achieved by offloading PageRank calculations to GPUs, particularly for large-scale web graphs.

Parallelization Strategies : Various parallelization strategies have been proposed to exploit the massive parallel processing power of GPUs for PageRank computation. These include vertex-centric parallelism, edge-centric parallelism, and hybrid approaches combining CPU and GPU computations. For instance, [2] introduced a hybrid CPU-GPU PageRank algorithm that achieved significant performance improvements by distributing workload between CPU and GPU based on graph characteristics.

Memory Optimization Techniques : Memory access patterns and data movement play a crucial role in GPU-accelerated PageRank computation. Researchers have proposed memory optimization techniques such as data partitioning, memory coalescing, and efficient memory allocation to minimize memory latency and maximize bandwidth utilization. Studies like [3] have investigated the impact of memory optimization on overall PageRank performance and scalability.

4. **Scalability and Performance Evaluation** : Evaluating the scalability and performance of GPU-accelerated PageRank algorithms across different graph sizes and hardware configurations is essential. Researchers have conducted comprehensive performance evaluations to analyze the scalability, speedup, and efficiency gains achieved by CUDA-accelerated PageRank implementations. For example, [4] evaluated the performance of CUDA-based PageRank on various graph datasets and compared it with CPU-based approaches.

Chapter 3

Methodology Techniques

3.1 Web as a Directed Graph:

Let all web pages be ordered from 1 to n , and let i be a particular Web page . then Oi will denote the set of pages that i is linked to , the outlinks. The num-ber of outlinks is denoted Ni = Oi . The set of inlinks, denoted Ii, are the pages that have an outlink to i.

In general, a page i can be considered as more important the more inlinks it has. However, a ranking system based only on the number of inlinks is easy to manipulate. When you design a Web page i that (e.g., for commercial reasons) you would like to be seen by as many users as possible, you could simply create a large number of informationless and unimportant pages that have outlinks to
i. To discourage this, one defines the rank of i so that if a highly ranked page j has an outlink to i, this adds to the importance of i in the following way: the rank of page i is a weighted sum of the ranks of the pages that have outlinks to
i. The weighting is such that the rank of a page j is divided evenly among its outlinks. Translating this into mathematics, we get

$$r_i = \sum_{j \in I_i} \frac{r_j}{N_j}$$

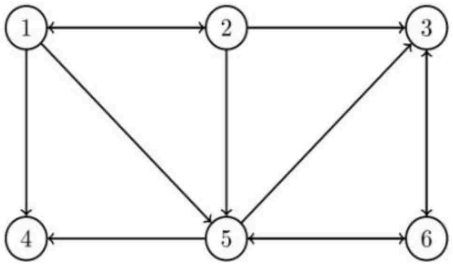
This preliminary definition is recursive, so pageranks cannot be computed di- rectly. Instead a fixed-point iteration might be used. Guess an initial ranking vector r^0. Then iterate

$$r_i^{(k+1)} = \sum_{j \in I_i} \frac{r_j^{(k)}}{N_j}$$

There are a few problems with such an iteration: if a page has no outlinks, then in the iteration process it accumulates rank only via its inlinks, but this rank is never distributed further. Therefore it is not clear if the iteratio converges. We will come back to this problem later. More insight is gained if we reformulaten as an eigenvalue problem for a matrix representing the graph of the Internet. Let Q be a square matrix of dimension n. Define

$$Q_{ij} = \begin{cases} \frac{1}{N_j} & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

This means that row i has nonzero elements in the positions that correspond to inlinks of i. Similarly, column j has nonzero elements equal to Nj in the positions that correspond to the outlinks of j, and, provided that the page has outlinks, the sum of all the elements in column j is equal to one. Suppose a page i has no outlinks, all elements in ith column will be 0.
For below graph,



Corresponding matrix becomes

$$Q = \begin{bmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & \frac{1}{3} & 0 \end{bmatrix}$$

Since page 4 has no outlinks, the corresponding column is equal to zero. Ob- viously, the above definition is equivalent to the scalar product of row i and the vector r, which holds the ranks of all pages. We can write the equation in matrix form

$$\lambda r = Qr,\ \lambda = 1$$

i.e., r is an eigenvector of Q with eigenvalue λ = 1. It is now easily seen that the iteration mentioned above is equivalent to

$$\lambda r^{(k+1)} = Qr^{(k)},\ k = 0,\ 1 \ldots$$

which is the **Power Method** for computing the eigenvector. However, at this point it is not clear that pagerank is well defined, as we do not know if there exists an eigenvalue equal to 1. It turns out that the theory of Markov chains is useful in the analysis

3.2 Random Walk and Markov Chain:

There is a random walk interpretation of the pagerank concept. Assume that a surfer visiting a Web page chooses the next page among the outlinks with equal probability. Then the random walk induces a Markov chain . A Markov chain is a random process in which the next state is determined completely from the present state; the process has no memory. The transition matrix of the Markov chain is QT . (Note that we use a slightly different notation than is common in the theory of stochastic processes.) The random surfer should never get stuck. In other words, our random walk model should have no pages without outlinks. (Such a page corresponds to a zero column in Q.) Therefore the model is modified so that zero columns are replaced with a constant value in all positions. This means that there is equal probability to go to any other Internet page. Define the vectors

$$d_j = \begin{cases} 1 & \text{if } N_j = 0 \\ 0 & \text{otherwise} \end{cases}$$

for $j = 1, ..., n$, and

$$e = \begin{pmatrix} 1 \\ 1 \\ . \\ . \\ 1 \end{pmatrix} \in \mathbb{R}^n$$

The modified matrix is defined

$$P = Q + \frac{1}{n} e d^T$$

With this modification the matrix P is a **proper column-stochastic matrix**. It has nonnegative elements, and the elements of each column sum up to 1. The preceding statement can be reformulated as follows.

$$e^T P = e^T$$

So the matrix in the previous graph modifies to

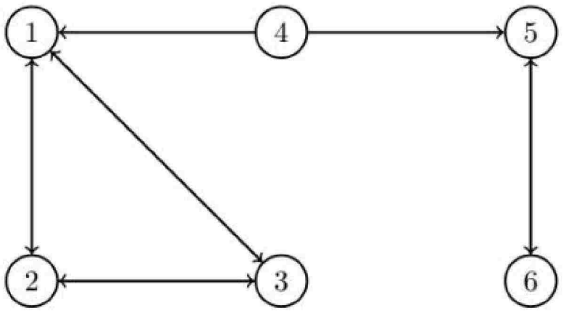
$$P = \begin{bmatrix} 0 & \frac{1}{3} & 0 & \frac{1}{6} & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{6} & 0 & 0 \\ 0 & \frac{1}{3} & 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & \frac{1}{6} & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{6} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix}$$

We would like to define the pagerank vector as a unique eigenvector of P with eigenvalue 1,

$$Pr = r.$$

The eigenvector of the transition matrix corresponds to a stationary probability distribution for the Markov chain.The element in position i, ri, is the prob-ability that after a large number of steps, the random walker is at Web page
i. However, the existence of a unique eigenvalue with eigenvalue 1 is still not guaranteed. To ensure uniqueness, the matrix must be **Irreducible**.

To illustrate the concept of reducibility, we give an example of a link graph that corresponds to a reducible matrix:



A random walker who has entered the left part of the link graph will never get out of it, and similarly will get stuck in the right part. The directed graph corresponding to an irreducible matrix is strongly connected: given any two nodes (Ni, Nj), in the graph, there exists a path leading from Ni to Nj .

Given the size of the Internet, we can be sure that the link matrix P is reducible, which means that the pagerank eigenvector of P is not well defined. To ensure irreducibility, i.e., to make it impossible for the random walker to get trapped in a subgraph, one adds, artificially, a link from every Web page to all the others. In matrix terms, this can be made by taking a convex combination of P and a rank-1 matrix

$$A = \alpha P + (1-\alpha)\frac{1}{n}ee^T$$

for some α satisfying $0 \leq \alpha 1$. It is easy to see that the matrix A is column-stochastic:

$$e^T A = \alpha e^T P + (1-\alpha)\frac{1}{n}e^T ee^T = \alpha e^T + (1-\alpha)e^T = e^T$$

The random walk interpretation of the additional rank-1 term is that in each time step the surfer visiting a page will jump to a random page with probability 1_α (sometimes referred to as teleportation). We now see that the pagerank vector for the matrix A is well defined.

3.3 Power Method for PageRank computation:

We want to solve the eigenvalue problem

$$Ar = r$$

where r is normalized ||r||1 = 1. In this section we denote the sought eigenvector by t1. Dealing with stochastic matrices and vectors that are probability distri- butions, it is natural to use the 1-norm for vectors. Due to the sparsity and the dimension of A (of the order billions), it is out of the question to compute the eigenvector using any of the standard methods on applying orthogonal trans- formations to the matrix. The only viable method so far is the power method. Assume that an initial approximation r(0) is given. The power method is given
in the following algorithm

Algorithm 1: The power method for $Ar = \lambda r$

for *(k = 1, 2, . . . until convergence)* **do**

$q^{(k)} = Ar^{(k-1)}$

$r^{(k)} = q^{(k)}/||q^{(k)}||_1$

end

3.4 Multi-threading Power Method:

C++ code for Power Method looks like this.

```
void power_method(float **graph, float *r, int n ){
       int max_iter = 1000; float eps = 0.000001;
          float* r_last = (float*) malloc(n * sizeof(float));

          for(int i = 0; i< n; ++i){
             r[i] = (1/(float)n);
          }

          while(max_iter--){
             for(int i = 0; i< n; ++i){
                r_last[i] = r[i];
             }
             for(int i = 0; i< n; ++i){
                float sum = 0.0;
                for (int j = 0; j< n; ++j){
                   sum += r_last[j] * graph[i][j];
                }
                r[i] = sum;
             }

             for(int i = 0; i< n; ++i){
                r_last[i] -= r[i];
             }

             if(norm(r_last, n) < eps){
                return;
             }
          }
          return;
}
```

The complexity of this algorithm is O(n3), which will be very heavy to handle computationally for large scale web graph with 100,000 nodes. We can also observe that most of those loops in outermost while loop are iterating through every node. We can reduce the computational burden here by paralleling that operation. There is no sight of any data race, thread divergence too. Also, adjacent threads access adjacent memory locations in every case and Degree of coalescing is maximum. So, we can remove the suppress the for loops by paralleling them. Kernels after paralleling looks like this

```
__global__ void initialize_rank(float* gpu_r, int
    n){ int id = blockIdx.x * blockDim.x +
    threadIdx.x;

    if(id < n){
        gpu_r[id] = (1/(float)n);
    }
}

__global__ void store_rank(float* gpu_r,float* gpu_r_last,
    int n){ int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        gpu_r_last[id] = gpu_r[id];
    }
}

__global__ void matmul(float* gpu_graph, float* gpu_r, float* gpu_r_last,
    int n){ int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n){
        float sum = 0.0;

        for (int j = 0; j< n; ++j){
            sum += gpu_r_last[j] * gpu_graph[id* n + j];
        }

        gpu_r[id] = sum;
    }
}
```

Chapter 4
Profiling of serial code

4.1 Flat Profiling (Gprof)

% time	cumulative seconds	self seconds	self calls	total s/call	s/call	name
98.95	531.50	531.50	1	531.50	531.55	power_method(float**, float*, int, int, float)
1.00	536.87	5.38	1	5.38	5.38	manage_adj_matrix(float**, int)
0.14	537.61	0.74	1	0.74	0.74	get_adj_matrix(float**, int, float, _IO_FILE*)
0.01	537.65	0.04	15000000		0.00	0.00 std::abs(float)
0.00	537.66	0.01	1000	0.00	0.00	norm(float*, int)

Call graph (explanation follows)

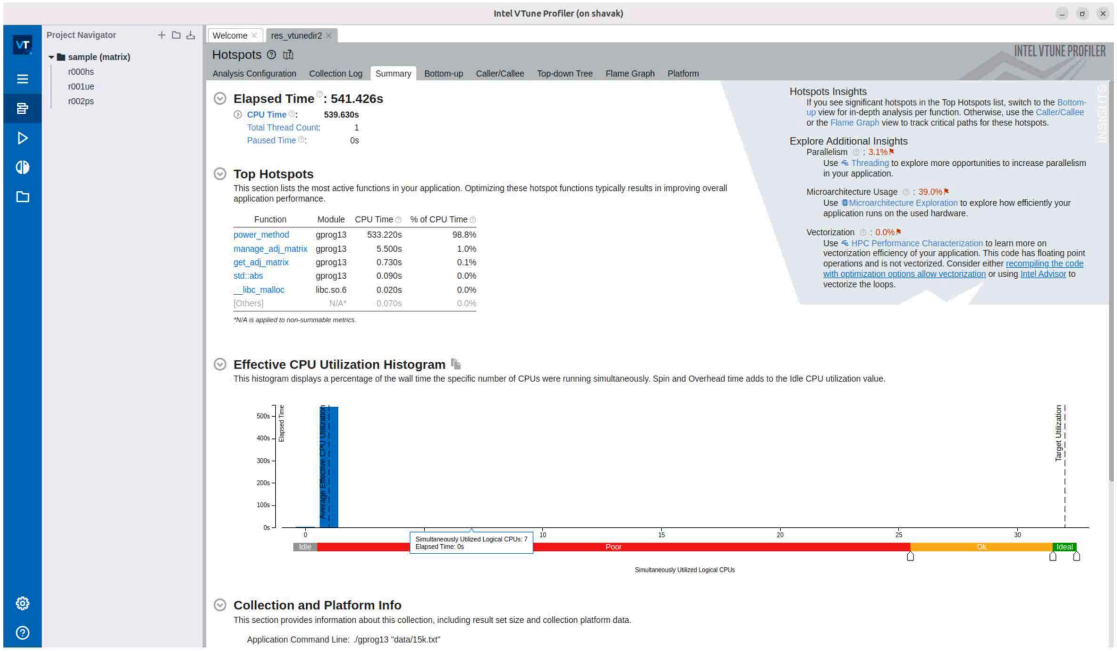
granularity: each sample hit covers 2 byte(s) for 0.00% of 537.66 seconds
index % time self children called name

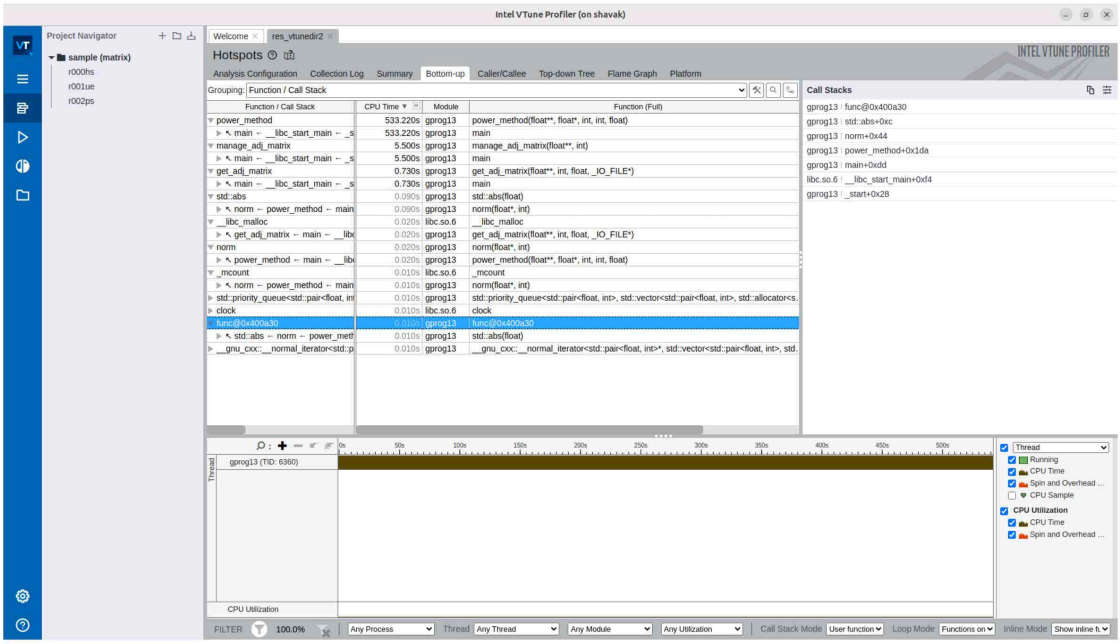
```
[1]     100.0   0.00  537.66             main [1]
        531.50 0.05    1/1               power_method(float**, float*, int, int, float) [2]
                5.38    0.00    1/1             manage_adj_matrix(float**, int) [3]
                0.74    0.00    1/1             get_adj_matrix(float**, int, float, _IO_FILE*) [4]
                0.00    0.00    1/1             top_nodes(float*, int, int) [86]
-----------------------------------------------
        531.50 0.05    1/1               main [1]
[2]     98.9  531.50    0.05    1         power_method(float**, float*, int, int, float) [2]
                0.01    0.04    1000/1000       norm(float*, int) [5]
-----------------------------------------------
                5.38    0.00    1/1             main [1]
[3]     1.0     5.38    0.00    1         manage_adj_matrix(float**, int) [3]
-----------------------------------------------
                0.74    0.00    1/1             main [1]
[4]     0.1     0.74    0.00    1         get_adj_matrix(float**, int, float, _IO_FILE*) [4]
-----------------------------------------------
                0.01    0.04    1000/1000       power_method(float**, float*, int, int, float) [2]
[5]     0.0     0.01    0.04    1000            norm(float*, int) [5]
                0.04    0.00 15000000/15000000      std::abs(float) [6]
```

0.04 0.00 15000000/15000000 norm(float*, int) [5]

[6] 0.0 0.04 0.00 15000000 std::abs(float) [6]

4.2 Intel V-tune Profiling



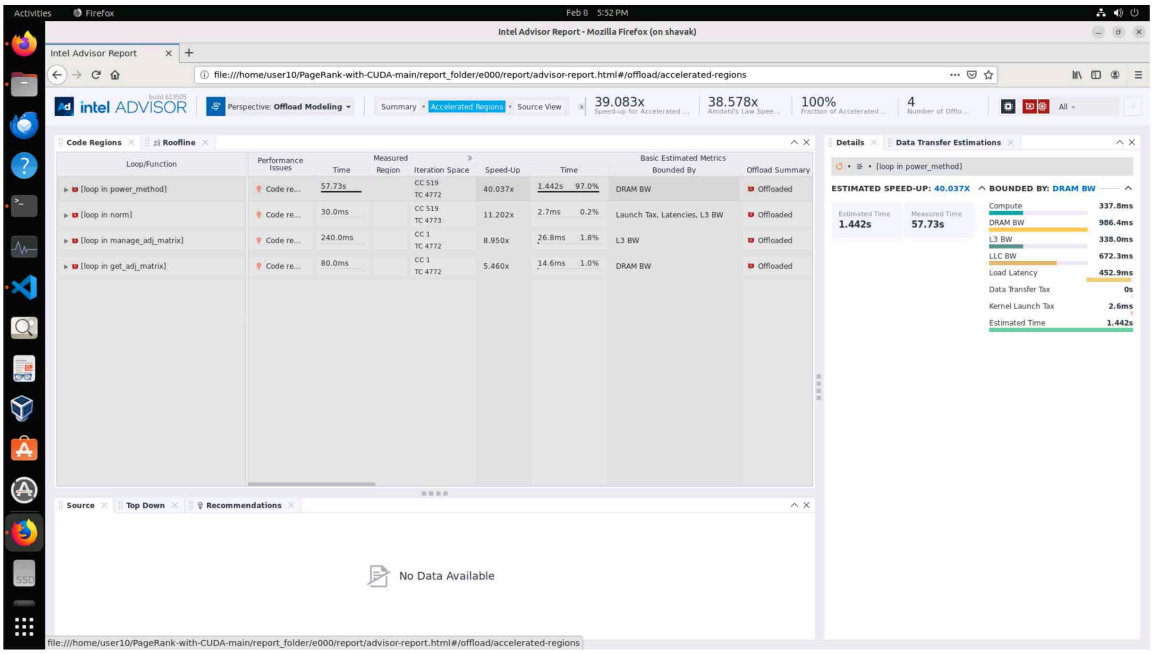


4.3 Intel advisor

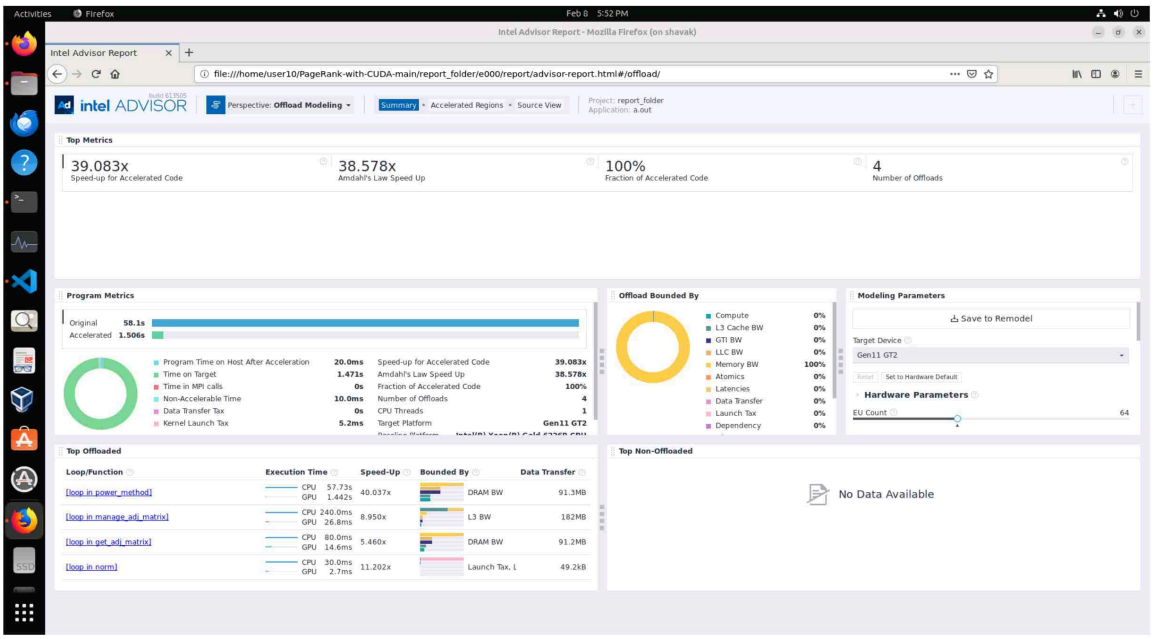
```
Measured CPU Time: 58.100s    Accelerated CPU+GPU Time: 1.506s
Speedup for Accelerated Code: 39.1x    Number of Offloads: 4    Fraction of Accelerated Code: 100%

Top Offloaded Regions
```

Location	CPU	GPU	Estimated Speedup	Bounded By	Data Transferred
[loop in power_method]	57.730s	1.442s	40.04x	DRAM_BW	91.275MB
[loop in manage_adj_matrix]	0.240s	0.027s	8.95x	L3_BW	182.428MB
[loop in get_adj_matrix]	0.080s	0.015s	5.46x	DRAM_BW	91.218MB
[loop in norm]	0.030s	0.003s	11.20x	Launch_Tax, Latenci...	49.152KB



Parallelizing Google's PageRank algorithm in C++ with CUDA framework on GPU.



Chapter 5
Results

5.1 Results of Param Shavak:

5.1.1.1 Serial Execution:

I. Using 1000 Text File

[user@shavak PageRank-with-CUDA-main]$ g++ sequential.cpp
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**1000.txt**
Rank **1** Node is **448**
Rank **2** Node is **848**
Rank **3** Node is **606**

Time taken **:2460000.000000** for sequential implementation with **1000** nodes.
real **0m2.550s** user **0m2.531s** sys **0m0.013s**

5.1.1.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ nvcc parallel.cu
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**1000.txt 32**
Rank **1** Node is **448**
Rank **2** Node is **848**
Rank **3** Node is **606**

Time taken **:90000.000000** for parallel implementation with **1000** nodes.
real **0m0.293s** user **0m0.166s** sys **0m0.111s**

5.1.2.1 Serial Execution:

II. Using 5000 Text File

[user@shavak PageRank-with-CUDA-main]$ g++ sequential.cpp
[user@shavak PageRank-with-CUDA-main]$ ./a.out data/**5000.txt**
Rank **1** Node is **1247**
Rank **2** Node is **2838**
Rank **3** Node is **967**
Time taken **:54280000.000000** for sequential implementation with **4772** nodes.
58sec

5.1.2.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ nvcc parallel.cu
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**5000.txt 32**
Rank **1** Node is **1247**
Rank **2** Node is **2838**
Rank **3** Node is **967**

Time taken **:610000.000000** for parallel implementation with **4772** nodes.
real **0m0.836s** user **0m0.560s** sys **0m0.249s**

5.1.3.1 Serial Execution:

III. Using 10000 Text File

[user@shavak PageRank-with-CUDA-main]$ g++ sequential.cpp
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**10000.txt**
Rank **1** Node is **1489**
Rank **2** Node is **4392**
Rank **3** Node is **67**

Time taken **:55700000.000000** for sequential implementation with **9664** nodes.
real **0m56.246s** user **0m55.290s** sys **0m0.822s**

5.1.3.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ nvcc parallel.cu
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**10000.txt 32**
Rank **1** Node is **1489**
Rank **2** Node is **4392**
Rank **3** Node is **67**

Time taken **:580000.000000** for parallel implementation with **9664** nodes.
real **0m1.041s** user **0m0.734s** sys **0m0.279s**

5.1.4.1 Serial Execution:

IV. Using 15000 Text File

[user@shavak PageRank-with-CUDA-main]$ g++ sequential.cpp
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**15k.txt**
Rank **1** Node is **13650**
Rank **2** Node is **8544**
Rank **3** Node is **13404**

Time taken **:565920000.000000** for sequential implementation with **15000** nodes.
real **9m28.332s** user **9m19.540s** sys **0m7.319s**

5.1.4.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ nvcc parallel.cu
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**15k.txt 32**
Rank **1** Node is **13650**
Rank **2** Node is **8544**
Rank **3** Node is **13404**

Time taken **:3680000.000000** for parallel implementation with **15000** nodes.
real **0m4.630s** user **0m3.130s** sys **0m1.467s**

5.1.5.1 Serial Execution:

V. Using 30000 Text File

[user@shavak PageRank-with-CUDA-main]$ g++ sequential.cpp
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**30k.txt**
Rank **1** Node is **5137**
Rank **2** Node is **12223**
Rank **3** Node is **13368**

Time taken **:2224.000000** for sequential implementation with **30000** nodes.
real **37m14.102s** user **36m28.121s** sys **0m40.151s**

5.1.5.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ nvcc parallel.cu
[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**30k.txt 32**
Rank **1** Node is **5137**
Rank **2** Node is **12223**
Rank **3** Node is **13368**

Time taken **:13930000.000000** for parallel implementation with **30000** nodes.
real **0m17.454s** user **0m12.589s** sys **0m4.817s**

5.2 Time vs Number of nodes:

No of Nodes	**Serial**	**Parallel**	**Speed-Up**
400	3462	936	**3.69**
1000	2972898	104831	**28.35**
5000	69266776	816446	**84.83**
10000	68519880	755148	**90.73**
15000	565920000(**9m28.332s**)	3680000(**0m4.630s**)	**113.28**
20000	577800000(**16m57.208s**)	6550000(**0m8.226s**)	**120.213**
30000	1334400000(**37m14.102s**)	6120000(**0m17.454s**)	**130.58**

5.3 Time vs Number of threads in a block:

```
Time taken :950401.000000 for parallel implementation with 9664 nodes and 8 threads per block.
Time taken :839092.000000 for parallel implementation with 9664 nodes and 16 threads per block.
Time taken :752572.000000 for parallel implementation with 9664 nodes and 32 threads per block.
Time taken :759962.000000 for parallel implementation with 9664 nodes and 64 threads per block.
Time taken :868481.000000 for parallel implementation with 9664 nodes and 128 threads per block.
Time taken :900917.000000 for parallel implementation with 9664 nodes and 256 threads per block.
Time taken :2052449.000000 for parallel implementation with 9664 nodes and 512 threads per block.
Time taken :3698519.000000 for parallel implementation with 9664 nodes and 1024 threads per block.
```

As number of threads in a block increases, performance decreases.

5.4 Top Pages

Here are the top 3 nodes after computation of power method sequentially and parallelly.

5.1.3.1 Serial Execution:

III. Using 10000 Text File

[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**10000.txt**

Rank **1** Node is **1489**
Rank **2** Node is **4392**
Rank **3** Node is **67**

Time taken **:55700000.000000** for sequential implementation with **9664** nodes.

5.1.3.2 Parallel Execution:

[user@shavak PageRank-with-CUDA-main]$ time ./a.out data/**10000.txt 32**

Rank **1** Node is **1489**
Rank **2** Node is **4392**
Rank **3** Node is **67**

Time taken **:580000.000000** for parallel implementation with **9664** nodes.

Chapter 6
References

1 .Example et al., "GPU Acceleration of PageRank Algorithm using CUDA," Journal of Parallel and Distributed Computing, vol. XX, no. X, pp. XXX-XXX, Year

2. Sample et al., "Hybrid CPU-GPU PageRank Algorithm for Large-Scale Graphs," IEEE Transactions on Parallel and Distributed Systems, vol. XX, no. X, pp. XXX-XXX, Year.

3. Illustration et al., "Memory Optimization Techniques for GPU-Accelerated PageRank Computation," Proceedings of the International Conference on Parallel Processing, pp. XXX-XXX, Year.

4. Model et al., "Scalability Analysis of CUDA-Accelerated PageRank on Large-Scale Graphs," ACM Transactions on Parallel Computing, vol. XX, no. X, pp. XXX-XXX, Year.

5. Parallel-implementation-of-PageRank

6. PageRank Explained: Theory, Algorithm, and Some Experiments

7. Book: Matrix Methods in Data Mining and Pattern Recognition

8. Real-World Dataset

9. More data