COMMUNITY DETECTION ON GPU: A COMPREHENSIVE ANALYSIS, UNIFIED MEMORY ENHANCEMENT, AND MEMORY ACCESS OPTIMIZATION

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Emre DİNÇER

December 2023
İZMİR

We approve the thesis of **Emre DİNÇER**

Examining Committee Members:

8 December 2023

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my thesis advisor, Dr. Işıl Öz, for her invaluable guidance and persistent support throughout the entire research process. Dr. Işıl Öz's insightful feedback has been instrumental in shaping the trajectory of this thesis. Honestly, I wouldn't have completed my research and this thesis without her support. I am truly fortunate to have had the opportunity to work under her mentorship, and I extend my sincere appreciation for her patience and exceptional guidance. Her profound impact on my academic and professional development will be a lasting influence, and I am profoundly thankful for that.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) due to the project's partnerships at FTGPGPU (Hardware Fault Tolerance Analysis for General Purpose Graphics Processing Units (GPGPU) Applications, Grant No: 119E011) and RAPPROX (Resource-Aware Compiler Design for Approximate Computing Techniques in General Purpose Graphics Processing Units (GPGPU) Applications, Grant No: 122E395).

ABSTRACT

COMMUNITY DETECTION ON GPU: A COMPREHENSIVE ANALYSIS, UNIFIED MEMORY ENHANCEMENT, AND MEMORY ACCESS OPTIMIZATION

Recent years have experienced a slowdown in the development of traditional systems that use only the Central Processing Unit (CPU). However, significant progress has been made in the development of heterogeneous systems utilizing not only the CPU but also the Graphics Processing Unit (GPU). NVIDIA, one of the GPU manufacturers, through its CUDA platform, has increased the interest of many researchers in heterogeneous systems by providing a means to program GPUs more easily. The ease of application development provided by the CUDA platform and the performance gains offered by these heterogeneous systems have encouraged many researchers to develop algorithms and applications that operate on these systems. One such algorithm that is frequently used in data analysis is the community detection algorithm. Although there are applications that implement this algorithm to run on GPUs, and while these applications work efficiently for many datasets, they either fail to work or experience significant performance loss for large datasets that exceed the GPU's memory capacity.

In this thesis, we analyzed Rundemanen, which is one of the community detection applications running on GPU. We also made enhancements that enable Rundemanen to process datasets larger than the GPU's memory capacity by utilizing CUDA's Unified Memory. Lastly, we tested various optimization methods to use Unified Memory more efficiently. By using our memory-access advises, in comparison to the naive version, we obtained up to 62x and 8x performance gain with artificial oversubscription scenarios and for datasets that already do not fit into GPU memory, respectively.

ÖZET

GRAFİK İŞLEMCİ BİRİMİ TABANLI TOPLULUK TESPİTİ: KAPSAMLI ANALİZ, BİRLEŞİK BELLEK DESTEĞİ VE BELLEK ERİŞİM OPTIMİZASYONU

Son yıllarda, yalnızca Merkezi İşlem Birimi'ni (MİB) kullanan geleneksel sistemlerin gelişiminde bir yavaşlama yaşanmış ancak sadece MİB'yi değil aynı zamanda Grafik İşlem Birimi'ni (GİB) de kullanan heterojen sistemlerin gelişiminde önemli ilerlemeler kaydedilmiştir. GİB üreticilerinden biri olan NVIDIA, kendi geliştirdiği CUDA platformu aracılığıyla araştırmacıların GİB'leri daha kolay programlamalarına olanak tanıyarak, heterojen sistemlere olan ilgiyi artırmıştır. CUDA platformu tarafından sağlanan uygulama geliştirme kolaylığı ve bu heterojen sistemler tarafından sunulan performans artışları, birçok araştırmacıyı bu sistemlerde çalışan algoritmalar ve uygulamalar geliştirmeye teşvik etmiştir. Veri analizinde sıkça kullanılan bu tür bir algoritma, topluluk tespiti algoritmasıdır. Bu algoritmaya dayalı uygulamaların GİB'de çalışmasını gerçekleyen uygulamalar bulunsa da, bu uygulamalar birçok veri kümesi için verimli çalışırken, MİB'nin bellek kapasitesini aşan büyük veri kümeleri için çalışmamakta veya önemli performans kayıpları yaşamaktadır.

Bu çalışmada, topluluk tespiti algoritmasını GİB üzerinde uygulayan uygulamalardan biri olan Rundemanen'i analiz ettik. Ayrıca, bu uygulamanın CUDA'nın Unified Memory özelliğini kullanarak MİB'nin bellek kapasitesini aşan veri kümelerini işlemesini sağlayan iyileştirmeler yaptık. Son olarak, Unified Memory'yi daha verimli hale getirmek için çeşitli optimizasyon yöntemlerini sunduk. Bellek erişim şeklini değiştirerek, aşırı abonelik (oversubscription) senaryoları için basit (naive) sürümle karşılaştırıldığında 62 kata kadar performans artışı elde ettik ve GİB belleğine sığmayan veri kümeleri için ise 8 kata kadar performans artışı sağladık.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

1.1. Introduction

Gordon Moore, co-founder of Intel, made a remarkable observation in 1965 that, every year, the number of transistors on a chip doubles without extra costs (Moore 2006). He revised it in 1975 by restating it would happen every two years rather than one. Later, this observation started to be known as Moore's Law. Moore's Law inevitably has implications for technological advancements in semiconductor industries. While the computational power of scalar single-core processors was increasing, array (vector) processors came into existence to be used heavily in scientific simulations, signal processing, etc. Later, commercial single-core processors (Peleg and Weiser 1996) began to include support vector operations similar to array processors with limited features.

In the early 2000s, people realized that increasing transistor counts or clock frequency was not applicable anymore due to physical limitations and others. As a physical limitation, the Power Wall (Kuroda 2001), a limiting factor caused by the need for excessive power, thereby leading to unmanageable heat dissipation, became an obstacle to producing computationally faster processors. The industry ceased shrinking transistors and raising clock frequencies to increase processor throughput. Instead, it began adding more components to enable processors to handle multiple tasks simultaneously. This transition started the multi-core era (Venu 2011). The first multi-core processor, POWER4, was introduced by IBM in 2001, and then Intel began to come into existence in this field around 2005 with its dual-core processor along with AMD (Alseqyani and Almutairi 2023). Then, Intel and AMD began to increase the core count, leading to the emergence of many-core processors, such as Intel's Xeon-Phi and NVIDIA's and AMD's GPUGPUs.

Although both multi-core and many-core processors enhance the computational capability of the system, their architectures and usage areas differ. Unlike multi-core processors, which are primarily employed for general-purpose computing, many-core processors have gained popularity in handling highly parallel workloads, such as scientific simulations, data analytics, and artificial intelligence. This popularity is attributed to the nature of tasks that can be divided into numerous smaller tasks capable of running almost independently in parallel. NVIDIA's GPGPUs are among the most widely utilized many-core processors, and their popularity can be ascribed to the developer-friendly programming environment offered by NVIDIA, known as CUDA (Compute Unified Device Architecture).

Most algorithms have benefited from the computational power of NVIDIA's GPGPUs, and their running time performance has improved; however, there are some exceptions, such as graph algorithms, due to their irregular memory access pattern (Burtscher, Nasre, and Pingali 2012). Additionally, because GPGPUs have limited memory space, dataset sizes have become so large that they do not fit into the memory. To solve this, in 2014, NVIDIA introduced Unified Memory, enabling developers to utilize all the available system memory by GPGPUs. However, this approach has performance drawbacks unless developers understand the application's memory usage pattern and take action using Unified Memory API.

In this thesis, the study focuses on one of the graph applications: community detection. Utilizing NVIDIA's Unified Memory, the goal is to improve the application's performance for datasets that do not fit into the GPGPU's memory.

1.2. Thesis Organization

We have organized this thesis into five distinct chapters:

- In the first chapter, we present an introduction to the thesis, explain the contribution, and mention the thesis organization.
- The second chapter, titled "Background," explains the functioning and internals of GPGPUs, Unified Memory, graph representations, and the community detection algorithm, particularly focusing on the Louvain implementation.
- Chapter three delves into an in-depth analysis of one specific Louvain implementor, Rundemanen. We classify data structures and group them based on their usage, elucidate source code modifications for Unified Memory utilization, and explain how we collect memory accesses and page faults. This chapter also outlines the methodology for creating artificial oversubscription scenarios and applying memory advises utilized in the experiments.
- In chapter four, besides introducing the datasets employed in this study and presenting meta-information about them, we systematically collect and analyze each dataset's memory accesses and page faults and, considering the results, recommend suitable memory advice hints. Lastly, we test the performance of the memory advises over the naive Unified Memory version of the application.
- In chapter five, we draw conclusions from the results obtained through the experiments.

1.3. Contributions

In this thesis, we select the best application of the community detection algorithm, namely Rundemanen, running on the GPU and enhance its capability to handle datasets exceeding GPU memory capacity by leveraging Unified Memory. We analyze the implementation in detail and suggest a method to collect memory access patterns. After pinpointing the hotspot on memory accesses and analyzing the page faults, we recommend several memory advises that alter the data access methodology. Applying this advises results in significant performance improvements over the naive version. Remarkably, we achieve performance gains of up to 69x and 8x with artificial oversubscription scenarios and datasets surpassing GPU memory limits, respectively.

CHAPTER 2

BACKGROUND

2.1. GPUGPUs

GPGPUs, which stands for General-Purpose Graphics Processing Units, are processing units that are basically used for graphic rendering as well as some applications requiring heavy computations in fields such as scientific computing, physical simulations, image processing, machine learning, and AI. Initially, they were called GPUs since they were used only for graphics rendering. Later, developers and researchers began investigating the possibility of utilizing GPUs for general-purpose computing applications. Finally, the vendors started to enhance their products so that they directly support general-purpose computing; as a result, they took the name of GPGPUs (Owens et al. 2007).

2.2. CUDA Platform

Compute Unified Device Architecture (CUDA) is a revolutionary platform introduced by NVIDIA in 2007 for programming GPUs produced by NVIDIA. It offers a unique execution model along with a programming model called SIMT. In addition to supporting multiple programming languages (C, C++, Fortran), it provides various libraries and middlewares for use with these languages (Wikipedia 2023).

2.2.1. Examining an Architecture

Figure 2.1.[1] shows the block diagram of Fermi microarchitecture released by NVIDIA in 2010. It is composed of several Streaming Multiprocessors(SMs), which are the main processing blocks with special instruction cache, L1 cache, integer and floating

1. Credits https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute _architecture_whitepaper.pdf

point arithmetic units, load/store units, and more as shown in Figure 2.2. ??. Newer architectures have similar components with extended features.

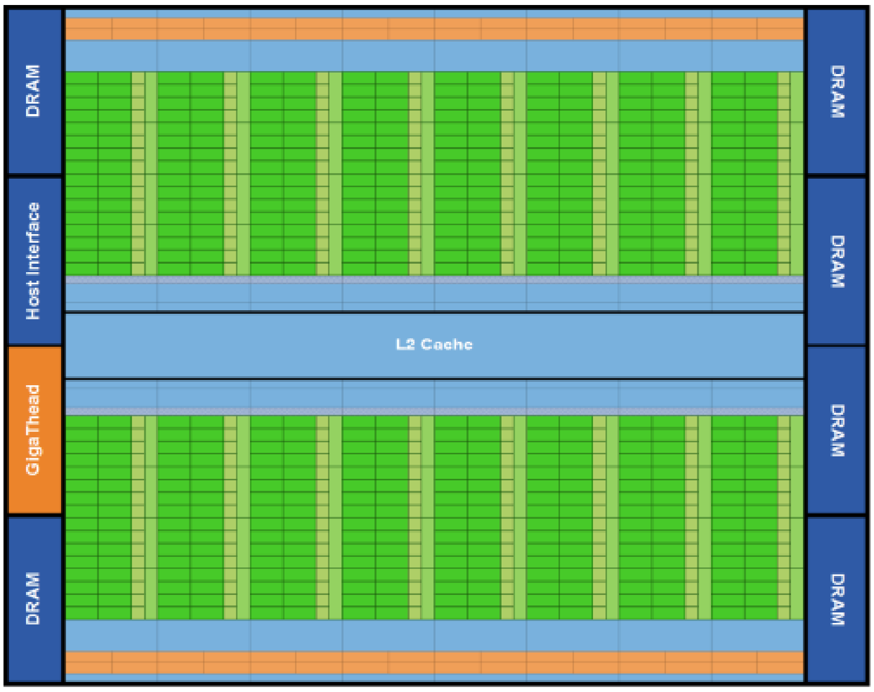


Figure 2.1. The block diagram of Fermi microarchitecture.

2.2.2. Execution Model

One prominent CUDA feature is its simple yet powerful execution model. The execution model guides us in understanding how GPUs operate at the hardware level.

Tasks, which are called CUDA kernels, assigned to a GPU are executed by CUDA threads, and each thread belongs to a block. Each of these blocks is assigned to an SM (Streaming Multiprocessor) at runtime. Although the smallest processing unit is a thread, threads are executed in groups. These groups are called warps, and each warp contains 32 threads (John Cheng 2014).

Every SM contains one or more Warp Schedulers that manage the warps and hold information about the warps currently assigned to that SM (Streaming Multiprocessor). If a warp is ready to execute, meaning that the resources it needs are available and it is not waiting on any memory operations, it is selected to run based on its priority status.

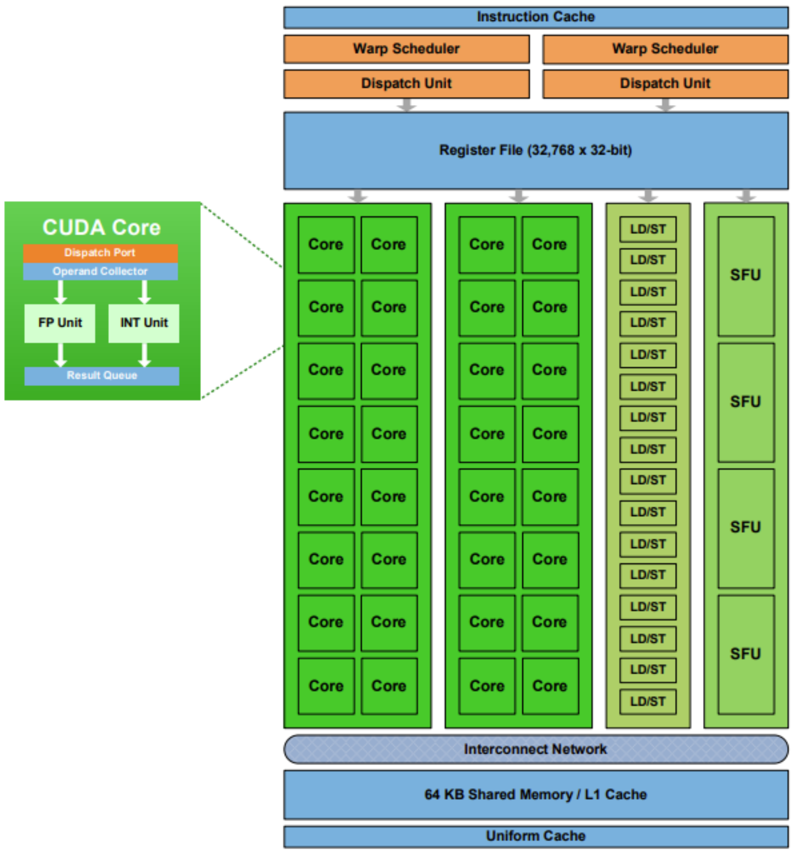


Figure 2.2. The block diagram of an SM(Streaming Multiprocessor).

Subsequently, the dispatch unit fetches and decodes the instructions for the warp and then issues the instructions. If not enough cores are available when the warp is executed, the warp is divided into subsets of threads which are executed sequentially.

If threads within the same warp encounter a branching instruction, and some of these threads take this branch while others continue with a different branch, a warp divergence(branch divergence) situation occurs. In this case, one of these divergent thread groups is executed while the other group waits in an idle state and is executed later (Han and Abdelrahman 2011).

2.2.3. Memory Model

Besides the execution model, CUDA provides a memory model that shows how memory operations are applied. CUDA has multiple logical memory spaces (NVIDIA 2023a); these memory areas are described below.

- Register File: This is a space where local variables belonging to threads and parameters of kernels are stored.

- Local Memory: This is the memory space used when threads require more local space.
- L1 Data Cache: This is a Global and Local memory cache area.
- Shared Memory: This space allows communication between threads within a thread block.
- L2 Cache: This is a cache area for Global, Local, Constant, Texture and Surface Memories.
- Constant Memory: This is a low-capacity area used to store read-only data. It has its own cache.
- Global Memory: This is the memory area with the highest capacity.
- Zero-Copy Memory: This is the memory area of system memory that is directly accessible by GPUs.

In addition to these, there are L0 and L1 Instruction caches and Texture and Surface Memory areas. The memory areas mentioned may not have a physical counterpart or be sharing a physical space with another memory area.

2.2.4. Programming Model

Before CUDA, although researchers and developers used GPUs for applications beyond graphics, the solutions had to be implemented as if they were graphics applications. This made using GPUs very challenging. This situation was resolved thanks to CUDA's programming model, making it easy to program GPUs for various application domains.

Programs written in CUDA operate according to the SIMT (Single Instruction, Multiple Threads) programming model (John Cheng 2014). Although this programming model resembles SIMD (Single Instruction, Multiple Data), in SIMT, each thread has its own PC (Program Counter) (in Volta and later architectures), resources, and execution path. Threads come together to form thread blocks, and thread blocks come together to form grids.

2.2.4.1. Kernels

The function containing the code we want to run on the GPU is a CUDA kernel. The number of thread blocks that would execute CUDA kernels and the number of threads within each thread block can be configured. The configuration parameters may vary depending on the code to be executed and the architecture. Setting these parameters correctly is crucial as they significantly impact performance.

Listing 2.1. A simple kernel applying the square function over the array's elements.

```
__global__ void square(int* data, int size) {
    int globalId = blockIdx.x * blockDim.x + threadIdx.x;
    if(globalId >= size) {
        return;
    }
    data[globalId] = data[globalId] * data[globalId];
}
```

Listing 2.1. shows a simple CUDA kernel. This kernel calculates the square of the elements of the given input array and writes the results back to that array. Although every thread executes the same code, the value of the *globalId* variable inside each thread is different. This is because each thread belongs to a block owning a unique ID, and each thread has its own unique ID within the block.

Listing 2.2. A simple main function executing a kernel.

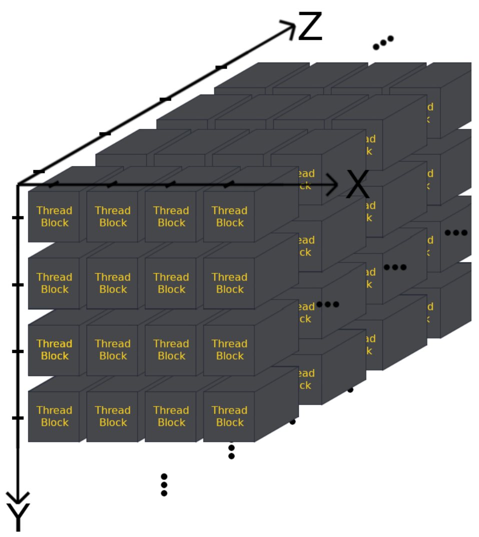
```
...
int block_size = 128;
int grid_size = (n_elems + block_size - 1) / block_size;
square<<<grid_size, block_size>>>(data, n_elems);
cudaDeviceSynchronize();
...
```

Listing 2.2. shows how the square kernel is configured and executed within a simple main code. We provide the kernel launch parameters between <<< and >>>, which in this case are *block_size* and *grid_size*.

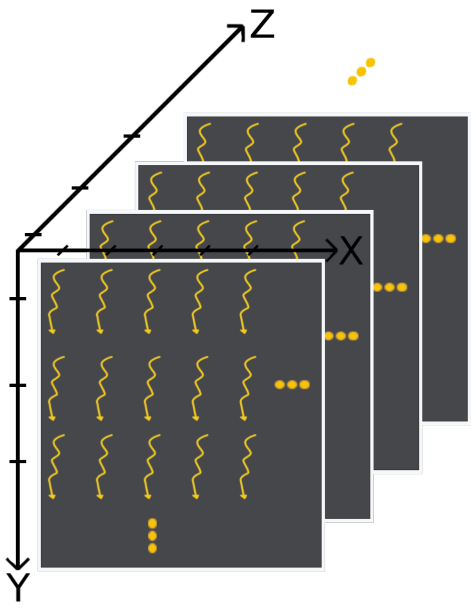
2.2.4.2. Grids & Blocks

Threads come together to form thread blocks, and thread blocks, in turn, come together to create grids. At runtime, a grid is created for each kernel. Blocks belonging to a grid can be assigned to any SM, and each completes its lifetime in that SM. *block_size* in Listing 2.2. specifies how many threads will run in each block. Since we want to apply the operation on all elements of the array, we calculated *grid_size* considering the array size to be processed (*n_elems*).

A grid can be 1D, 2D, or 3D, represented by an index at a point $(X[,Y[,Z]])$ that corresponds to a distinct thread block. Figure 2.3.a provides a pictorial representation of a grid. Within kernels, programmers can determine which thread block the current



(a) A pictorial view of a grid.



(b) A pictorial view of a block.

Figure 2.3. A pictorial view of Grid & Block.

thread belongs to using the built-in variables *blockIdx*.x, *blockIdx*.y, and *blockIdx*.z. The dimensions of a grid can be obtained using *gridDim*.x, *gridDim*.y, and *gridDim*.z. If the grid is 1D, then *gridDim*.y, *gridDim*.z are set to 1, while *blockIdx*.y, and *blockIdx*.z are set to 0. In the case of a 2D grid, *gridDim*.z is 1 and *blockIdx*.z is 0. To calculate the global block ID, the following formula can be used:

$$\begin{aligned} globalBlockId = {} & blockIdx.z \times (gridDim.x \times gridDim.y) \\ & + blockIdx.y \times gridDim.x \\ & + blockIdx.x \end{aligned} \tag{2.1}$$

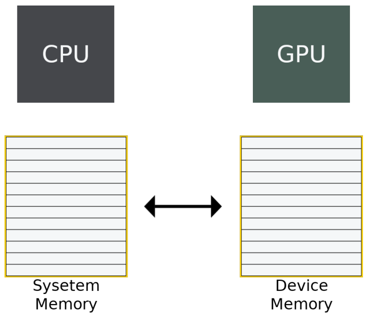
A block can be 1D, 2D, or 3D, with each point corresponding to a thread, as illustrated in Figure 2.3.. Within kernels, CUDA provides three built-in variables for indexing a thread: *threadIdx*.x, *threadIdx*.y, and *threadIdx*.z. Additionally, the dimensions of a block can be determined using the built-in variables *blockDim*.x, *blockDim*.y, and *blockDim*.z. The 1D and 2D conditions applicable to grids also apply to blocks. To calculate the global thread ID, the following formula is used:

$$\begin{aligned} globalThreadId = {} & threadIdx.z \times (blockDim.x \times blockDim.y) \\ & + threadIdx.y \times blockDim.x \\ & + threadIdx.x \end{aligned} \tag{2.2}$$

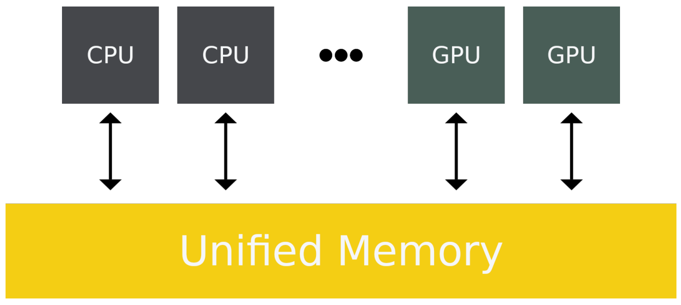
When initiating kernels, providing grid and block dimensions as integers results in a 1D grid, with each block being 1D as well. To form a 2D or 3D grid, we can use *dim3* type objects while launching a kernel. The constructor of dim3 can take up to three parameters, with each argument corresponding to the dimensions' lengths of X, Y, and Z.

2.2.5. Unified Memory

Unified Memory is a memory abstraction introduced by NVIDIA with CUDA 6. It enables the host and device to share a single address space, thereby making memory management very simple for developers. It also supports memory oversubscription, allowing devices to use more memory than it has. Without Unified Memory, the programmers must manually move data back and forth between the host and device, which requires so much effort to manage.



(a) without Unified Memory



(b) with Unified Memory

Figure 2.4. Pictorial view of Unified Memory.

Without Unified Memory, when we want to process data on the device, we first need to allocate space on the device using *cudaMalloc* API call and then copy the data from the host to this allocated space on the device using *cudaMempy* API call. When the device finishes its tasks, we also need to copy the data back from the device to the host memory. However, with Unified Memory, there is no need for both separate allocation and copying. A simple kernel execution with and without Unified Memory is shown in Listing 2.3. and 2.4., respectively.

Listing 2.3. A sample code without Unified Memory.

```
...
constexpr size_t SIZE = 1000000;
int* host_array = (int*)malloc(SIZE * sizeof(int));
initialize(host_array);

int* device_array;
cudaMalloc(&device_array, SIZE * sizeof(int));
```

```
cudaMemcpy(device_array, host_array,
           SIZE * sizeof(int),
           cudaMemcpyHostToDevice);

my_kernel<<<...,...>>>(device_array, SIZE);
cudaDeviceSynchronize();

cudaMemcpy(host_array, device_array,
           SIZE * sizeof(int),
           cudaMemcpyDeviceToHost);
int total = 0;
for(size_t i = 0; i < SIZE; i++){
    total += host_array[i];
}
cudaFree(device_array);
free(host_array);
...
```

Listing 2.4. A sample code with Unified Memory.

```
constexpr size_t SIZE = 1000000;
int* array;
cudaMallocManaged(&array, SIZE * sizeof(int));

my_kernel<<<...,...>>>(array, SIZE);
cudaDeviceSynchronize();

int total = 0;
for(size_t i = 0; i < SIZE; i++){
    total += array[i];
}
cudaFree(array);
...
```

2.2.5.1. Internals

The smallest building block of Unified Memory is a page typically 4096B in size. When the CPU or GPU tries to access data, the runtime first checks whether the corresponding page resides in its memory. If not, the page is migrated to the accessed side's memory. This mechanism is called on-demand page migration. Besides, sometimes, runtime speculatively prefetch a page before accessing it to utilize the bandwidth effectively and increase the performance.

2.2.5.2. Memory Advises

We can control the behavior of the UM page handling mechanism thanks to the UM Advises. CUDA offers an API call *cudaMemAdvise* that we can apply to a specific memory address range. Here are the advises that we can apply:

- cudaMemAdviseSetReadMostly: informs the runtime that the address range will be used frequently for read operations, and write operations are not expected for the specific processor. Thus, a copy of the pages corresponding to the specified address range is created by the processor where the operation is performed, and read operations are carried out on this copy. When a write operation is performed, copies on all other processors become invalidated, excluding the one where the operation was performed.
- cudaMemAdviseSetPreferredLocation: is used to determine the runtime's data placement. If the processor seeking access has direct access to the preferred location, the operation is executed without moving the page; otherwise, the runtime migrates the page. Nonetheless, it aids the runtime in preventing memory thrashing at the preferred location.
- cudaMemAdviseSetAccessedBy: allows a specific processor to access the data without the need to transfer the page that contains it, thanks to remote mapping. When the pages are migrated to the memory of another processor, the mapping is automatically reestablished.

2.2.5.3. Oversubscription Support

Oversubscription is a situation where the processor uses more memory than its memory space. For CPUs, this is done by migrating unused data to swap space residing on a disk. When it comes to GPUs, they use the host's memory as a swap area when it needs more memory space than is currently available. The runtime evicts a page to the host memory when the memory is full, according to an eviction policy. Because of the eviction, a page might be migrated back and forth, if not at every access, every few accesses. This is called memory thrashing and could lead to dramatic performance degradation.

2.3. Graphs

Many entities in nature and the relationships among these entities can be represented in the form of a graph: people and their friendship connections in social networks, subatomic particles in the field of physics and their interactions, molecules and the chemical bonds between them in the field of chemistry, etc. Entities are represented as vertices in the graph, relationships as edges, and the data associated with the relationship are represented as the edge weights (Shi et al. 2018).

2.3.1. Graph Representation

In programming, graph datasets can be represented in various ways. Non-negative integer IDs are typically assigned to vertices and used. Starting vertex IDs from 0 facilitates indexing operations on data structures. We present the following representations' pictorial view in Figure 2.5..

2.3.1.1. Adjacency Matrix

In this representation, a two-dimensional matrix is created. Each entry in the matrix determines the presence of an edge from the vertex corresponding to the row ID (tail) to the vertex corresponding to the column ID (head). The entry's value is considered the edge weight, and entries with a value of 0 indicate no edge. If the graph is undirected, the matrix is symmetric.

2.3.1.2. Adjacency List

An adjacency array (or list) is created for each vertex, and these arrays are stored in an array that can be accessed using the vertex ID. The head vertex of each edge incident to the vertex is added to the adjacency list of that vertex. If the graph is weighted, these arrays can be stored as vertex-weight pairs or a separate adjacency weight array can be created for the weights.

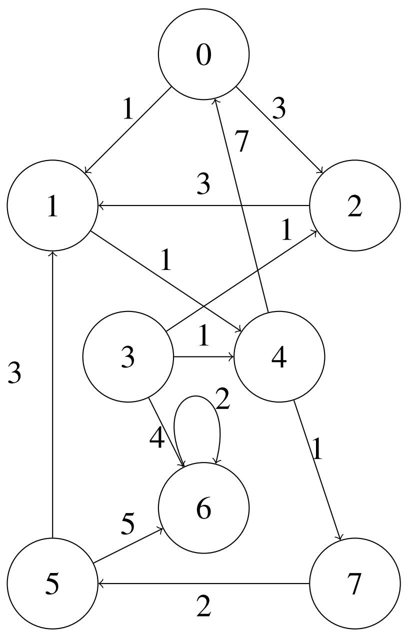
2.3.1.3. Edge List

Edges are stored in a list in pairs or triples if there is weight. If the graph is undirected, instead of keeping two entries for each edge ($headID$, $tailID$ and $tailID$, $headID$), storing a single entry for the edge is sufficient.

2.3.1.4. Compressed Formats

If the graph is sparse, with many 0s in the adjacency matrix, storing it in the matrix would be inefficient in terms of memory space. Even in cases with no edges, the entry still holds a value of 0, leading to unnecessary memory usage. Therefore, several compressed representations were developed: Dictionary of keys (DOK), Coordinate list (COO), and Compressed Sparse Row (CSR).

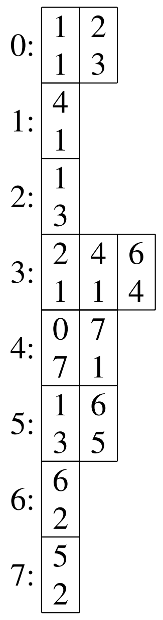
Among these formats, CSR is widely used. CSR is composed of three arrays: *indices*, *links*, and *weights*. The *indices* array determines vertices' neighbors' start and end indices. The *links* array stores the IDs of neighboring vertices, and if the graph is weighted, the *weights* array stores the edge weights. To find where a vertex's neighbors start and end in the *links* array, the data corresponding to the vertex ID in the *indices* array and the data corresponding to the next vertex ID (one more than the current vertex ID) are used.



(a) Graph

	0	1	2	3	4	5	6	7
0	0	1	3	0	0	0	0	0
1	0	0	0	0	1	0	0	0
2	0	3	0	0	0	0	0	0
3	0	0	1	0	1	0	4	0
4	7	0	0	0	0	0	0	1
5	0	3	0	0	0	0	5	0
6	0	0	0	0	0	0	2	0
7	0	0	0	0	0	2	0	0

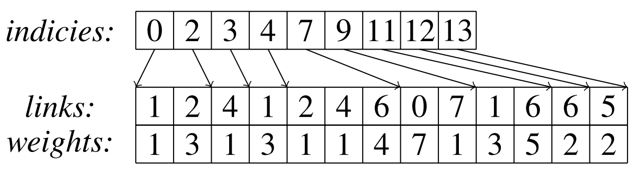
(b) Adjacency Matrix



(c) Adjacency List

```
0,1,1
0,2,3
1,4,1
2,1,3
3,2,1
3,4,1
3,6,4
4,0,7
4,7,1
5,1,3
5,6,5
6,6,2
7,5,2
```

(d) Edge List



(e) Compressed Sparse Row(CSR)

Figure 2.5. Various representations of an example graph.

2.4. Community Detection

Community Detection is a type of graph analysis that groups nodes within a graph based on their interactions with each other or according to specific properties. These groups, known as communities, help to reveal the patterns of interaction both within and between them. Several community detection techniques have been proposed: Partitioning, Statistical Inference, Dynamical methods, Deep Learning methods, Optimization-based, and others (Su et al. 2022). Within the Optimization-based methods, the algorithms try to maximize a predefined quality function such as Modularity (Newman and Girvan 2004) being heavily used. Fast Greedy Algorithm (Newman 2004), Louvain (Blondel et al. 2008), ITS (Lü and Huang 2009), and some genetic algorithms (Liu, Yang, and Liu 2016; Tasgin, Herdagdelen, and Bingol 2006) are those trying to optimize Modularity. Throughout this work, we will be examining Louvain due to its popularity. Louvain uses Modularity optimization to find communities.

2.4.1. Modularity

Modularity (Newman and Girvan 2004) is a metric used in network science that quantifies how well-defined the communities are in a network. Well-defined communities refer to the situation in which the vertices in each community are more densely connected with each other than the other vertices outside of their community.

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \text{ (Blondel et al. 2008)}$$

where:

- Q: Modularity
- $A_{i,j}$: weight of the edge between vertex i and vertex j
- k_i, k_j: the sum of the weights of the edges incident to vertex i and vertex j, respectively
- c_i, c_j: the communities of vertex i and vertex j respectively
- $\delta(c_i, c_j)$: 1 if $c_i = c_j$, else 0
- m: $\frac{1}{2} \sum_{i,j} A_{i,j}$

2.4.2. Louvain

Louvain (Blondel et al. 2008) is a greedy method that assigns each vertex to a community, maximizing the overall Modularity, and then creates a new graph in which the communities become new vertices. These phases are the Vertex Movement and the Aggregation (see Figure 2.6.), respectively. It applies these operations until the gain in Modularity drops below a certain threshold. Louvain method was initially developed for running on CPUs, and then several applications and libraries running on GPUs emerged. These applications use Louvain as a base, and each tries to optimize Louvain for GPUs. Rundemanen (Naim et al. 2017), cuGraph (RAPIDS.ai 2022), and cuVite (Gawande et al. 2022) are examples of such applications. cuGraph is a CUDA library developed by RAPIDS.ai. While cuVite makes Louvain applicable for multi-node systems, it can also be used for single-node systems. On the other hand, Rundemanen is developed for single-node systems, based on *grappolo* (Mahantesh Halappanavar and Ghosh 2020) that parallelizes the Louvain method for CPUs. Although these applications are built upon the Louvain method, they have employed certain heuristics to optimize their performance specifically for GPUs.

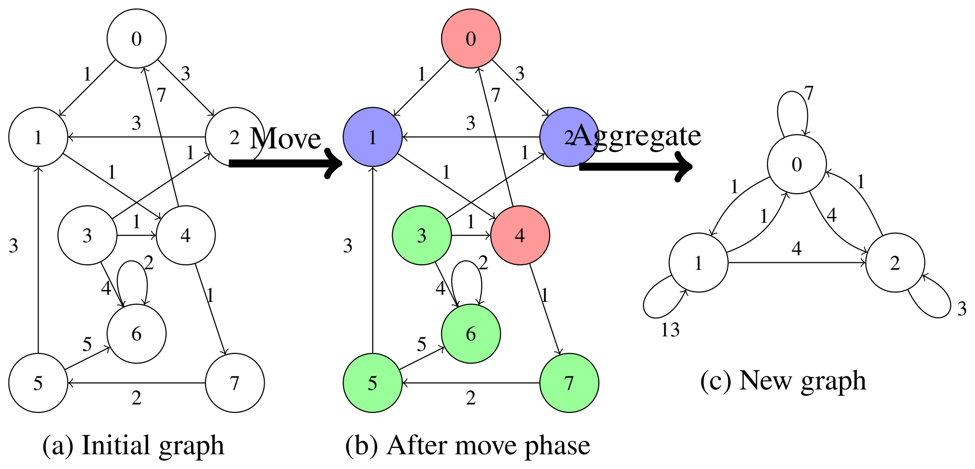


Figure 2.6. Louvain algorithm phases on an example graph.

Table 2.1. compares the performance of these applications on a Pascal GPU (Quadro P4000) across various datasets, which are subsequently employed in the experiments. While cuGraph and cuVite exhibit superior Modularity optimization for certain datasets, the runtime analysis reveals that Rundemanen significantly outperforms both cuGraph and cuVite. Additionally, cuGraph and cuVite face challenges, either due to excessive memory requirements surpassing available resources or quality levels reached

Table 2.1. Running times and qualities of the community detection applications collected in the PASCAL environment.

Dataset	**Rundemanen**		**cuGraph**		**cuVite**	
	Time(s.)	**Quality**	**Time(s.)**	**Quality**	**Time(s.)**	**Quality**
audikw_1[2]	0.751	0.621	17.46	0.953	44.52	0.924
soc-LiveJournal1[3]	2.901	0.724	40.47	0.716	45.58	0.725
Long_Coup_dt6[4]	0.979	0.912	17.50	0.998	18.73	0.995
dielFilterV3real[5]	0.624	0.547	-	-	20.38	0.938
cage15[6]	1.517	0.840	67.57	0.860	-	-
rgg_n_2_24_s0[7]	3.432	0.992	-	-	-	-
kron_g500-logn21[8]	1.484	0.042	76.88	0.035	-	-
uk-2002[9]	3.910	0.941	-	-	-	-

that are not in valid ranges, preventing proper execution on all datasets. Considering all these, choosing Rundemanen as the community detection application to analyze seems reasonable; therefore, in this thesis, we choose Rundemanen as the base application.

2. (Mayer 2004; Davis and Hu 2011)
3. (Leskovec and Krevl 2014; Davis and Hu 2011)
4. (Janna, Ferronato, and Gambolati 2012; Ferronato, Bergamaschi, and Gambolati 2010; Bergamaschi, Ferronato, and Gambolati 2008, 2007; Davis and Hu 2011)
5. (Dziekonski, Lamecki, and Mrozowski 2011; Davis and Hu 2011)
6. (Heukelum, Barkema, and Bisseling 2002; Davis and Hu 2011)
7. (Holtgrewe, Sanders, and Schulz 2010; Bader et al. 2014; Davis and Hu 2011)
8. (Bader et al. 2014; Davis and Hu 2011)
9. (Boldi and Vigna 2004; Boldi et al. 2011; Boldi et al. 2004; Davis and Hu 2011)

CHAPTER 3

DATA-ACCESS AWARE COMMUNITY DETECTION

In this thesis, we analyze Rundemanen and make enhancements that enable Rundemanen to process datasets larger than the Graphics Processing Unit's memory capacity by utilizing CUDA's Unified Memory. Also, to improve Unified Memory performance, we propose several memory-accessing hints on data structures.

3.1. Analysis of Rundemanen

Rundemanen is an implementation of the Louvain method, which is designed to be performant on GPUs. It utilizes NVIDIA's *thrust* (NVIDIA 2023b) library for trivial tasks like scan, reduce, gather, for-each, transform, sort, copy, and count. Besides, it uses thrust vectors that internally manage objects' (arrays) lifetime. For custom kernels, it employs a virtual warp-centric programming model (Hong et al. 2011) to mitigate workload imbalance problem, and uses hash tables for faster accesses.

Rundemanen encompasses two consecutive phases executed repeatedly until the Modularity gain between iterations falls below a certain threshold. These phases are of the Louvain method: Vertex Movement (moveVertices) and Aggregation (aggregate). The pseudocode for Rundemanen is outlined in *Algorithm 1*.

3.1.1. Initialization

The graph data is read from a file and then converted to Compressed Sparse Row (CSR) representation and stored in the device's global memory. The CSR representation comprises *indices*(*g.indices*), *links*(*g.links*), and *weights*(*g.weights*). *indices*, *links* are types of *thrust::device_vector<int>* and *thrust::device_vector<unsigned int>* respectively; and, *weights* is the type of *thrust::device_vector<float>*. Their lengths are $|V| + 1$, $|E|$, and $|E|$. The lifetime of a CSR object ends after the Aggregate phase, and it is replaced with the aggregated graph's CSR.

Along with the graph data, Rundemanen reads some pre-computed prime numbers

Algorithm 1 Rundemanen

```
Require: $CSR, primes$
1:  $threshold \leftarrow 0.000001$
2:  $threshold2 \leftarrow 0.01$
3:  while $true$ do
4:      $(vertToComm, quality) \leftarrow$ MOVEVERTICES$(CSR, primes, threshold2)$
5:      if $quality - prevQuality <= threshold$ then
6:          if isLastRound then
7:              break
8:          end if
9:          $isLastRound \leftarrow true$
10:     end if
11:     $CSR \leftarrow$ AGGREGATE$(vertToComm, CSR, primes)$
12: end while
```

from a file. They are used to determine hash tables' capacities', and they are placed into a *devPrimes*(*thrust::device_vector<int>*). The vector remains unchanged until the app terminates.

The dataflow chart of the Initialization phase is shown in Figure 3.1. below.

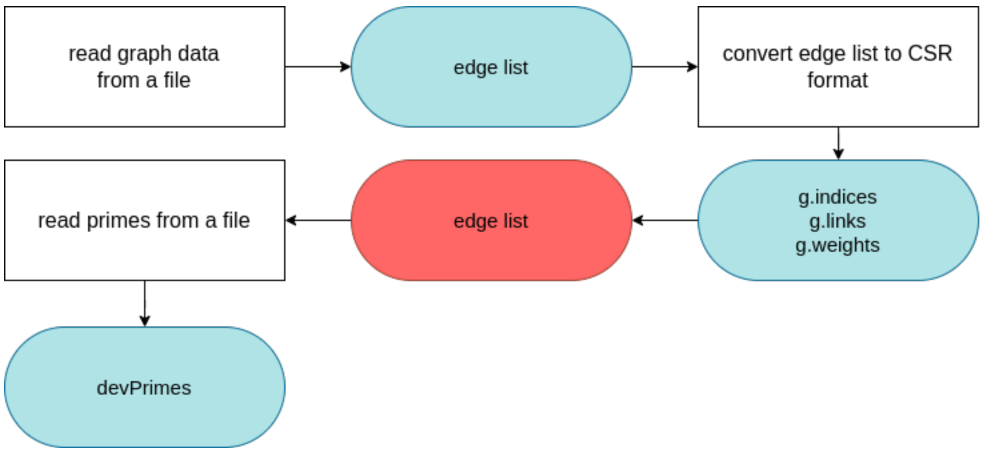


Figure 3.1. Initialization phase's flow chart.

Rounded light-aqua and light-red shapes show the construction and destruction of objects, respectively. Additionally, rectangular boxes show operations.

3.1.2. Vertex Movement

The Vertex Movement(moveVertices) phase (see Algorithm 2) involves multiple iterations where each vertex is assigned to a community to maximize the network's Modularity. First, vertices are grouped into buckets based on their degrees. Before the movement operations, each vertex is placed in a unique community. Each bucket of vertices is processed in a specific order. While applying the movement operation, each vertex is detached from its current community, and then the Modularity gain from this vertex to each of its neighboring communities, including the current one, is calculated. Subsequently, the vertex is placed in the community, leading to maximum Modularity gain. These steps are executed for each bucket, and then the total graph Modularity is evaluated; if the difference compared to the previous Modularity surpasses a specified threshold, the movement operations are rerun. If not, the vertex movement phase concludes.

Algorithm 2 moveVertices(Host)

```
Require: $CSR, primes, threshold$
 1: $verticesDegrees \leftarrow$ calculateVerticesDegrees($CSR$)
 2: ($bucket1to4, bucket5to8,$
        $bucket9to16, bucket17to32,$
        $bucket33to83, bucket84to318,$
        $bucket319toInf$) $\leftarrow$ bucketize($verticesDegrees$)
 3: $hashTables \leftarrow$ allocateHashTables($bucket319toInf, bucket84to318$)
 4: $graphTotalWeight \leftarrow$ calculateGraphTotalWeight($CSR$)
 5: $verticesWeights \leftarrow$ calculateVerticesWeights($CSR$)
 6: $communityInfo \leftarrow$ initializeCommunityInfo($CSR, verticesWeights$)
 7: while $numIters <= 1000$ do
 8:     moveBigDegVertices($bucket319toInf, hashTables, CSR,$
                           $communityInfo, primes, verticesWeights,$
                           $graphTotalWeight$)
 9:     moveSmallDegVertices($bucket5to8, CSR, communityInfo,$
                             $primes, verticesWeights, graphTotalWeight$)
10:     moveSmallDegVertices($bucket9to16, ...$)
11:     moveSmallDegVertices($bucket1to4, ...$)
12:     moveSmallDegVertices($bucket17to32, ...$)
13:     moveSmallDegVertices($bucket33to83, ...$)
14:     moveBigDegVertices($bucket84to318, hashTables, ...$)
15:     $modularity \leftarrow$ calculateModularity($communityInfo$)
16:     if modularity < previousModularity then
17:         break
18:     end if
19: end while
```

Figure 3.2. shows a dataflow chart of the Vertex Movement phase and the objects' lifetimes. Here is the explanation of the objects:

- *sizesOfNhoods* is a *thrust::device_vector<int>* of size $|V|$. It is used to bucketize the vertices. Elements at indices are the degrees of vertices.
- *g_next.indices* is a *thrust::device_vector<int>* of size $|V|$. Unlike its name, it stores the vertices of buckets such that the vertices of the buckets whose degree range is bigger are placed first.
- *g_next.links* is a *thrust::device_vector<unsigned int>* of size $|V|$. Though its name implies that it is for storing graph data, it is used as temporary storage.
- *globalHashTable* is a *thrust::device_vector<HashItem>*. It stores the entries of the hash tables, and its length equals two times the vertices' degrees sum. These vertices are from the first two buckets that hold the largest degree vertices, and if the number of vertices exceeds 90, the first 90 is considered. A HashItem comprises two fields: the target community as *int* and the aggregated weight as *float* from a vertex to the target community. Moreover, to store the beginning indices of the hash tables for these 90 vertices —at max—, a *thrust::device_vector<int>*, namely *hashTablePtrs*, is used.
- *wDegs* is a *thrust::device_vector<float>* of size $|V|$. It stores the weight of each vertex.
- *n2c_old*, *n2c*, and *n2c_new* are types of *thrust::device_vector<int>* of size $|V|$. They store the corresponding community for each vertex. *n2c_old* and *n2c_new* are used temporarily between kernels. *n2c* is also used in the Aggregate phase.
- *tot*, *tot_new* are types of *thrust::device_vector<float>* of size $|V|$. They store each community's weights equal to its vertices' aggregated weights.
- *in* is a *thrust::device_vector<float>* of size $|V|$. It stores each community's inner(self-loop) weights —, i.e. the total weights from each vertex to others inside the community.
- *cardinalityOfComms*, *cardinalityOfComms_new* are types of *thrust::device_vector<int>* of size $|V|$. They store each community's size.
- *result_array* is a *thrust::device_vecor<double>* of size $|V|$. It stores each community's contribution to the graph Modularity.

In Figure 3.2., we see several functions call. For the sake of brevity, we group actual functions in the source code into these. They internally use *thrust* functions and some custom kernels. However, the functions *lookAtNeighboringComms* and *neigh_comm* directly correspond to *moveBigDegVertices* and *moveSmallDegVertices*, respectively. These functions represent kernels executing on GPUs, optimizing the *Modularity*.

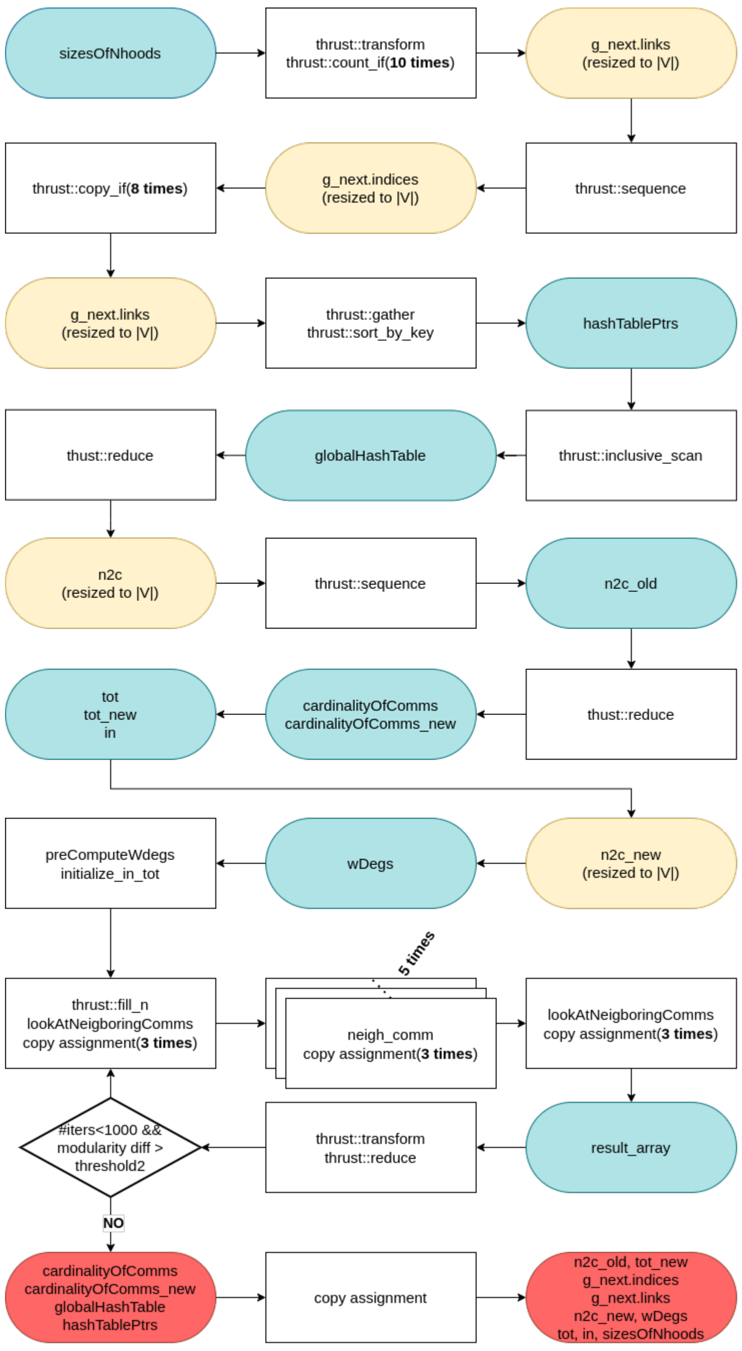

sizesOfNhoods
thrust::transform thrust::count_if(10 times)
g_next.links (resized to |V|)
thrust::sequence
g_next.indices (resized to |V|)
thrust::copy_if(8 times)
g_next.links (resized to |V|)
thrust::gather thrust::sort_by_key
hashTablePtrs
thrust::inclusive_scan
globalHashTable
thust::reduce
n2c (resized to |V|)
thrust::sequence
n2c_old
thust::reduce
cardinalityOfComms cardinalityOfComms_new
tot tot_new in
n2c_new (resized to |V|)
wDegs
preComputeWdegs initialize_in_tot
thrust::fill_n lookAtNeigboringComms copy assignment(3 times)
5 times
neigh_comm copy assignment(3 times)
lookAtNeigboringComms copy assignment(3 times)
result_array
thrust::transform thrust::reduce
#iters<1000 && modularity diff > threshold2
NO
cardinalityOfComms cardinalityOfComms_new globalHashTable hashTablePtrs
copy assignment
n2c_old, tot_new g_next.indices g_next.links n2c_new, wDegs tot, in, sizesOfNhoods

Figure 3.2. Vertex Movement phase's flow chart.

Rounded light-aqua, light-yellow, and light-red shapes show the construction, resizing, and destruction of objects, respectively. Additionally, rectangular boxes show function calls, and the diamond shape shows *if* statement.

Within *moveBigDegVertices*, each block processes only a vertex inside the bucket in a grid-stripe loop. The allocation of hash tables, contingent on the vertex's degree, occurs in either global or shared memory. Shared memory is selected if the degree falls below a specified threshold; otherwise, global memory is employed. Each thread within a block is responsible for computing the weight between the vertex and the community of an adjacent vertex. Given that multiple adjacencies of a vertex may be in the same community, the accumulated weight is aggregated in a hash table entry through the use of CUDA's *atomicAdd* API call. Following the completion of this calculation, the community leading to the optimal gain is identified within a block using warp-level shuffle functions. Subsequently, only one thread updates the relevant data structures.

In *moveBigDegVertices*, each warp is subdivided into sub-warps based on bucket processing, with each sub-warp assigned to process a specific vertex within a grid-stride loop. Shared memory is utilized for hash tables, and each thread in a sub-warp performs the same tasks as the threads in *moveBigDegVertices*.

3.1.3. Aggregation

Within The Aggregation(AGGREGATE) phase(see Algorithm 3), each non-empty community is selected and then renumbered. The weights of the edges connecting vertices within a community to vertices in other communities are aggregated. A new edge is created for every pair of communities if the aggregated weight is greater than 0. Moreover, the weights of the edges connecting the vertices to each other in a community are aggregated and assigned to the edge representing the self-loop. Once these computations are completed, the existing communities and edges are transformed into vertices and edges within the new graph structure.

Figure 3.3. shows a flowchart of the Aggregation phase and the objects' lifetimes. Here is the explanation of the objects:

- *n2c* is a *thrust::device_vector<int>* of size $|V|$. Its elements show which vertex belongs to which community. *n2c* is calculated in the previous phase.
- *renumber* is a *thrust::device_vector<int>* of size $|V|$. It is used to store which community's size is greater than 0.
- *n2c_new* is a *thrust::device_vector<int>* of size $|V|$. It is used as a mapping from a community to its new ID.
- *pos_ptr_of_new_comm* and *super_node_ptrs* are types of *thrust::device_vector<int>* of size $|new\ V|+1$. They store non-empty communities' sizes in a cumulative manner.

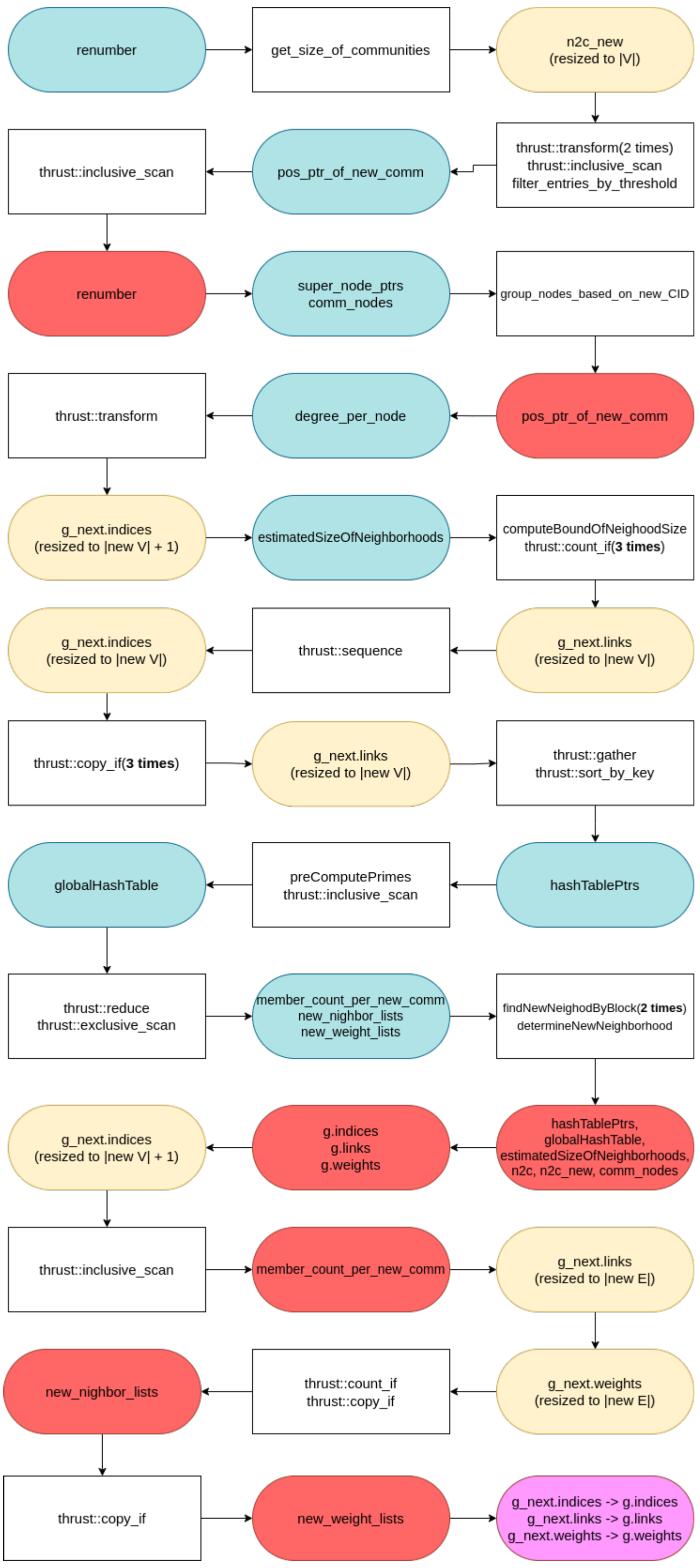

renumber
get_size_of_communities
n2c_new (resized to |V|)
thrust::transform(2 times) thrust::inclusive_scan filter_entries_by_threshold
pos_ptr_of_new_comm
thrust::inclusive_scan
renumber
super_node_ptrs comm_nodes
group_nodes_based_on_new_CID
pos_ptr_of_new_comm
degree_per_node
thrust::transform
g_next.indices (resized to |new V| + 1)
estimatedSizeOfNeighborhoods
computeBoundOfNeighoodSize thrust::count_if(3 times)
g_next.links (resized to |new V|)
thrust::sequence
g_next.indices (resized to |new V|)
thrust::copy_if(3 times)
g_next.links (resized to |new V|)
thrust::gather thrust::sort_by_key
hashTablePtrs
preComputePrimes thrust::inclusive_scan
globalHashTable
thrust::reduce thrust::exclusive_scan
member_count_per_new_comm new_nighbor_lists new_weight_lists
findNewNeighodByBlock(2 times) determineNewNeighborhood
hashTablePtrs, globalHashTable, estimatedSizeOfNeighborhoods, n2c, n2c_new, comm_nodes
g.indices g.links g.weights
g_next.indices (resized to |new V| + 1)
thrust::inclusive_scan
member_count_per_new_comm
g_next.links (resized to |new E|)
g_next.weights (resized to |new E|)
thrust::count_if thrust::copy_if
new_nighbor_lists
thrust::copy_if
new_weight_lists
g_next.indices -> g.indices g_next.links -> g.links g_next.weights -> g.weights

Figure 3.3. Aggregate phase's flow chart.

Rounded light-aqua, light-yellow, and light-red shapes show the construction, resizing, and destruction of objects, respectively. Additionally, rectangular boxes show function calls, the diamond shape shows *if* statement, and the rounded pink shape shows moving operations of the contents of the objects.

Algorithm 3 AGGREGATE(Host)

```
Require: $CSR, primes, communityInfo$
1: $renumberedCommsIDs \leftarrow$ RENUMBERCOMMUNITIES($communityInfo$)
2: $commSizes \leftarrow$ CALCULATECOMMUNITYSIZES($communityInfo$)
3: ($commBucket1to127$,
     $commBucket128to479$,
     $commBucket480toInf$) $\leftarrow$ GETVERTICES($commSizes$)
4: $hashTables \leftarrow$ ALLOCATEHASHTABLES($commBucket480toInf$,
                                          $commBucket128to479$)
5: $newGraph \leftarrow$ INITIALIZENEWGRAPH($commSizes, CSR$,
                                          $renumberedCommsIDs$)
6: AGGREGATEBIGCOMMS($commBucket480toInf, hashTables, primes$
                     $CSR, newGraph, commSizes$,
                     $renumberedCommsIDs$)
7: AGGREGATEBIGCOMMS($commBucket128to479, ...$)
8: AGGREGATESMALLCOMMS($commBucket1to127, primes$,
                       $CSR, newGraph, commSizes$,
                       $renumberedCommsIDs$)
9: $newGraphCSR \leftarrow$ CONVERTTOCSR($newGraph$)
10: return $newGraphCSR$
```

- *degree_per_node* is a *thrust::device_vector<int>* of size $|new\,V| + 1$. It stores non-empty communities' sizes.
- *comm_nodes* is a *thrust::device_vector<int>* of size $|V|$. It stores vertices such that the vertices in the same community are placed consecutively.
- *g_next.links* is a *thrust::device_vector<unsigned int>* of size $|new\,V|$. It is used as temporary storage until a point. Then it is resized to $|new\,E|$, and used as the aggregated(new) graph's CSR representation's *colind*.
- *g_next.indices* is a *thrust::device_vector<int>* of size $|new\,V|$. At first, it is used to store the communities of buckets such that the communities of the buckets whose degree range is bigger are placed first. Then it is resized to $|new\,V| + 1$, and used as the aggregated(new) graph's CSR representation's *rowptr*.
- *estimatedSizeOfNeighborhoods* is a *thrust::device_vector<int>* of size $|new\,V| + 1$. It stores the aggregated size of the number of vertices adjacent to all vertices in each community.
- *globalHashTable* is a *thrust::device_vector<HashItem>*. It stores the entries of the hash tables, and its length is equal to the communities' degrees sum —a community's degree is equal to the aggregated size of the number of vertices adjacent to all vertices in it. These communities are from the first two buckets that hold the largest degree communities, and if the number of communities exceeds 150, the first 150 are

considered. A HashItem comprises two fields: the target community in *int* and the aggregated weight in *float* from a community to the target community. Moreover, to store the beginning indices of the hash tables for these 150 communities —at max—, a *thrust::device_vector<int>*, namely *hashTablePtrs*, is used.

- *member_count_per_new_comm* is a *thrust::device_vector<unsigned int>* of size $|new\ V| + 1$. It stores the number of adjacent communities of each community. Later, this is used as the aggregated(new) graph's CSR representation's *rowptr*(g.indices).

- *new_nighbor_lists* and *new_weight_lists* are types of *thrust::device_vector<unsigned int>* and *thrust::device_vector<float>* respectively. Their lengths equal the sum of all elements of *estimatedSizeOfNeighborhoods*.

In Figure 3.3., we see several functions call. For the sake of brevity, we group actual functions in the source code into these. They internally use *thrust* functions and some custom kernels. However, the functions *aggregateBigComms* and *aggregateSmall-Comms* directly correspond to *findNewNeighodByBlock* and *determineNewNeighborhood*, respectively. These functions represent kernels executing on GPUs, constructing the new graph's data.

Within *findNewNeighodByBlock*, each block processes only a community in a grid-stripe loop. The allocation of hash tables, contingent on the number of vertices the community includes, occurs in either global or shared memory. Shared memory is selected if the size falls below a specified threshold; otherwise, global memory is employed. Each thread within a block is tasked with computing the weight between the vertex inside the community and its adjacent vertices' communities. Given that multiple vertices of a community may be in the same community, the accumulated weight is aggregated in a hash table entry through the use of CUDA's *atomicAdd* API call. Following this calculation's completion, the community's adjacent communities along with their weights are collected from the hash table and written into the new graph's data.

In *determineNewNeighborhood*, each warp is subdivided into sub-warps based on bucket processing, with each sub-warp assigned to process a specific community within a grid-stride loop. Shared memory is utilized for hash tables, and each thread in a sub-warp performs the same tasks as the threads in *findNewNeighodByBlock*.

3.2. Data Access-Aware Unified Memory for Rundemanen

In this section, we demonstrate the source code enhancements enabling support for large-sized graphs, detail the development of a memory access tracker, describe the implementation of data-access aware Unified Memory advises at variable granularity, illustrate our approach for gathering both spatial and temporal characteristics of program variables to assess their influence on execution performance and elucidate our method for monitoring both memory accesses and page faults in case of oversubscription.

3.2.1. Migration to Unified Memory

Our target implementation, Rundemanen, heavily uses NVIDIA's *thrust*'s (NVIDIA 2023b) *device_vector*, which is a type of object that wraps arrays with additional metadata and utility functions —they are like C++ *std::vector*. *thrust::device_vector* allocates memory on the device using *cudaMalloc*. If we want this allocation in Unified Memory, we can use *thrust::universal_vector*, which uses *cudaMallocManaged* internally.

Using *thrust::universal_vector* directly can create problems with certain memory advises, like setting *cudaSetPreferredLocation* to CPU. The issue arises during object construction and resizing. When an object of the *thrust::universal_vector* type is constructed or resized, an initialization function that initializes the elements on the GPU is called directly. Consequently, pages are created on the GPU before we can apply any memory advice. If we want data to be in a certain location like the CPU, we must migrate the pages, resulting in performance drawbacks.

One solution to this issue is to apply memory advice before the pages are created. By preventing the kernel responsible for initialization from being called immediately after construction or resizing and ensuring that this process occurs after applying memory advisories, the problem can be resolved. We have created a custom vector type, *my_vector*, which functions similarly to *thrust::universal_vector*. The only difference is that it doesn't perform automatic initialization, allowing us to apply memory advisories directly after construction and resizing operations. Manual initialization using *thrust::fill_n* is required after providing memory advisories (Note: Clearing is necessary just before the resize operation).

Listing 3.1. Custom vector implementation.

```
#include <thrust/universal_vector.h>
#include <thrust/execution_policy.h>

struct my_exec_pol
  : thrust::device_execution_policy<my_exec_pol>
{
};

template<typename T>
struct my_alloc
```

```
  : public thrust::universal_vector<T>::allocator_type
{
    using system_type = my_exec_pol;
    void system() {}
};

template<typename Iter, typename Size, typename T>
Iter uninitialized_fill_n(my_exec_pol,
                          Iter first,
                          Size,
                          const T&)
{
    // Do nothing
    return first;
}

template <typename T>
using my_vector = thrust::detail::vector_base<T,  my_alloc<T>>;
```

3.2.2. Other Modifications

We fixed compilation bugs, replaced deprecated CUDA warp shuffling functions, removed printing statements influencing the performance and the vectors allocated but not used, added warp synchronization statement, the absence of which causes incorrect results, added several prime numbers to the file including pre-computed primes and a bucket for 0-degree vertices to fix the runtime error, and call *shrint_to_fit* function after every *clear* function call whose object would not be used until next iteration to evaluate the memory requirement correctly.

We use C++ *std::swap* operation when we replace the old graph with the new one; after swapping, we reapply memory advises and *cudaMemPrefetchAsync* to migrate pages to the preferred location if *cudaMemAdviseSetPreferredLocation* is set.

Also, we edited the source codes to log operations like construction, resizing, swapping, clearing, and destruction applied to vectors. The log details vary by operation:

- Construction: operation type, operation time, object's name, start address, end address.
- Resize: operation type, operation time, object's name, new start address, new end address.
- Clear: operation type, operation time, object's name.
- Destruction: operation type, operation time, object's name.

- Swap: operation type, operation time, first object's name, seconds object's name.

3.2.3. Data Access Characteristics

For this thesis, we perform instrumentation on the application to collect memory accesses on Rundemanen. Using our instrumentation, we collect a set of metrics that specify the spatial and temporal characteristics of the object groups. While we collect memory access requests dynamically for general program execution, we also track page faults for oversubscribed scenarios.

3.2.4. Data Structures

We classify each vector based on its functionality in the implementation and group them into four categories:

- *Graph's CSR*: represents the CSR (Compressed Sparse Row) representation of the graph being processed. It is composed of: *g.indices*, *g.links*, and *g.weights*.
- *Hash Tables*: stores aggregated weight from vertices to communities. It is composed of: *globalHashTable*, *hashTablePtrs*, and *devPrimes*.
- *Community Information*: includes several objects storing community-related information, such as the weights and sizes of the communities and the community IDs of the vertices. It is composed of: *n2c*, *n2c_new*, *n2c_old*, *in*, *tot*, *tot_new*, *comm_nodes*, *cardinalityOfComms* and *cardinalityOfComms_new*.
- *Others*: includes objects used temporarily for buckets, new graph information, and other bookkeeping data. It is composed of: *g_next.indices*, *g_next.links*, *g_next.weights*, *sizesOfNhoods*, *wDegs*, *result_array*, *member_count_per_new_comm*, *new_nighbor_lists*, *new_weight_lists*, *pos_ptr_of_new_comm*, *super_node_ptrs*, *estimatedSizeOfNeighborhoods* and *renumber*.

3.2.4.1. Memory Access Tracking

To instrument the application, we add some functionality and data structures to handle memory accesses and write a Python script that modifies PTX source files.

Within *Memory Access Handler*, we define some variables holding essential data and implement a set of functions to perform data collection during program execution, as shown in Listing 3.2.. Specifically, we define six variables: a pointer for the buffer holding the memory addresses, a pointer for the buffer holding the access times, a pointer for the current time (Unix time in nanoseconds), the number of elements the buffers hold (size), the buffers' capacity, and the number of operations recorded. The buffers and the current time variable are allocated on the host memory with zero-copy mode, and the others are allocated on the device memory. The current time is updated every 0.1 seconds by a separate host thread. The variable holding the number of operations is used to skip some writing the buffers. To use the handler, we first initialize it in the main function and then periodically dump the buffers.

Listing 3.2. Variables and functions for memory access tracking.

```
constexpr int N_BUFFER_ELEMS = 1000000000;
__constant__ void** addresses_buffer_ptr;
__constant__ long*  times_buffer_ptr;
__device__   long   n_opers;
__constant__ long   capacity;
__device__   long   size;
__constant__ long*  current_time_ptr;

void update_timer() {
    // Updates the value current_time_ptr
    // every 0.1 seconds
}

void init_mem_acc_handler() {
    // Allocates space for the buffers
    // Initializes the variables
    // Starts a thread that runs update_timer()
}

void reset_buffers() {
    // Resets the buffers' content and the variables
}

void dump_memory_accesses() {
    // Writes the contents of the buffer into a file
    // Calls reset_buffers()
}
```

Since getting the information related to load and store operations is more practical, we create a Python script, *.ptx processor*(see Listing 3.3.), which modifies the target PTX code instead of CUDA kernel code modification. Our *.ptx processor* inserts the directives that include the buffers and variables' definitions into the target code. Then, it reads the original PTX code line by line and copies the line to the generated PTX code. If the line contains memory load and store operation on GPU global memory, the processor inserts our additional instructions that handle the memory access tracking with the address and time information.

Listing 3.3. The *ptx processor* Python Source Code.

```
import sys, re

def can_place_definitions(line):
     return ".address_size" in line

DEFINITIONS = """
.extern .const  .align 8 .u64 addresses_buffer_ptr;
.extern .const  .align 8 .u64 times_buffer_ptr;
.extern .const  .align 8 .u64 current_time_ptr;
.extern .const  .align 8 .u64 capacity;
.extern .global .align 8 .u64 size;
.extern .global .align 8 .u64 n_opers;
"""

INSTRUCTIONS = """
{
    .reg.pred               %myp;

    .reg.u64                %r_n_opers;
    atom.global.add.u64     %r_n_opers, [n_opers], 1;
    and.b64                 %r_n_opers, %r_n_opers, __SKIP_N_OPERS__;
    setp.ne.u64             %myp, %r_n_opers, 0;
    @%myp bra SKIP;

    .reg.u64                %r_capacity;
    ld.const.u64            %r_capacity, [capacity];
    .reg.u64                %r_current_index;
    atom.global.add.u64     %r_current_index, [size], 1;
    setp.ge.u64             %myp, %r_current_index, %r_capacity;
    @%myp bra SKIP;

    .reg.u64                %r_offset;
    mov.u64                 %r_offset, %r_current_index;
    shl.b64                 %r_offset, %r_offset, 3;

    .reg.u64                %r_page_addr;
    mov.u64                 %r_page_addr, __ADDRESS__;
    .reg.u64                %r_address_buffer_ptr;
    ld.const.u64            %r_address_buffer_ptr, [addresses_buffer_ptr];
    cvta.to.global.u64      %r_address_buffer_ptr, %r_address_buffer_ptr;
    add.u64                 %r_address_buffer_ptr, %r_address_buffer_ptr, %r_offset;
    st.global.u64           [%r_address_buffer_ptr], %r_page_addr;

    .reg.b64                %r_current_time_ptr;
    ld.const.u64            %r_current_time_ptr, [current_time_ptr];
    cvta.to.global.u64      %r_current_time_ptr, %r_current_time_ptr;
    .reg.u64                %r_current_time;
    ld.global.volatile.u64  %r_current_time, [%r_current_time_ptr];

    .reg.u64                %r_times_buffer_ptr;
    ld.const.u64            %r_times_buffer_ptr,    [times_buffer_ptr];
    cvta.to.global.u64      %r_times_buffer_ptr,    %r_times_buffer_ptr;
    add.u64                 %r_times_buffer_ptr,    %r_times_buffer_ptr, %r_offset;
    st.global.u64           [%r_times_buffer_ptr],  %r_current_time;
SKIP:
}
"""

ADDR_GROUP_NAME = "addr"
MEM_ACC_PATTERN = r"(ld|st|atom)\.(?!param|const|shared|local|volatile).*\[(?P<"\
                  + ADDR_GROUP_NAME\
                  + r">.*)\].*;"
SKIP_N_OPERS = 15 # must be (2^n - 1)

for file_path in sys.argv[1:]:
    file = open(file_path, "r")
    new_content = ""
    for line in file:
        new_content += line
        if can_place_definitions(line):
            new_content += DEFINITIONS
            continue
        match = re.search(MEM_ACC_PATTERN, line)
        if match:
            addr = match.group(ADDR_GROUP_NAME)
            new_content += INSTRUCTIONS.replace("__ADDRESS__", addr)\
                                       .replace("__SKIP_N_OPERS__", str(SKIP_N_OPERS))
    file.close()
    file = open(file_path, "w")
    file.write(new_content)
    file.close()
```

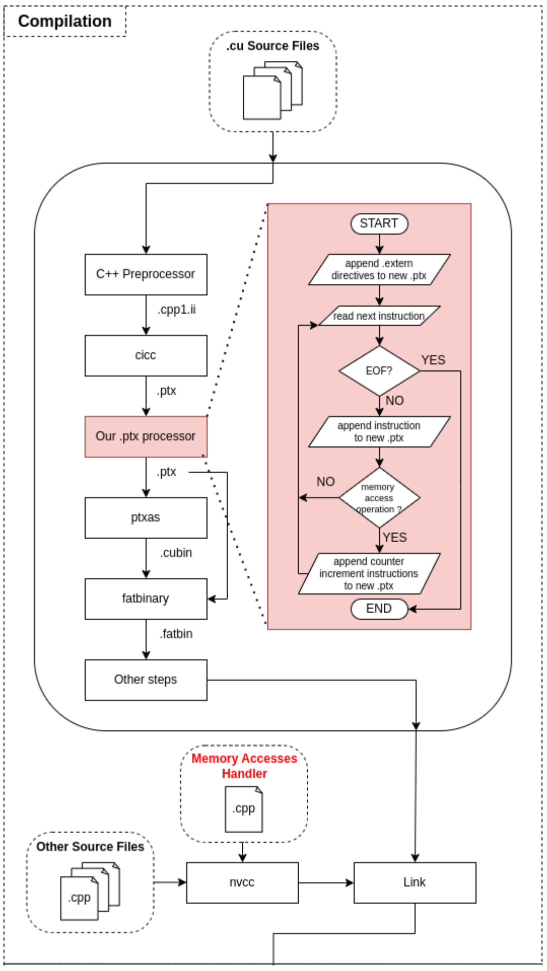
The application outputs memory access information, with each entry including the address and the time of the corresponding memory operation, along with the execution logs (see Section 3.2.2.). We use a Python script to determine which object's address range includes each address at its respective time.

The complete compilation and execution phases are described in Figure 3.4.. In the Compilation Phase (a), we manually apply the operations that the nvcc compiler uses internally to intercept the process and insert instrumentation-related code into the .ptx file. After adding instrumentation code, we complete the remaining compilation operations, and finally, run the linker to link all object files.

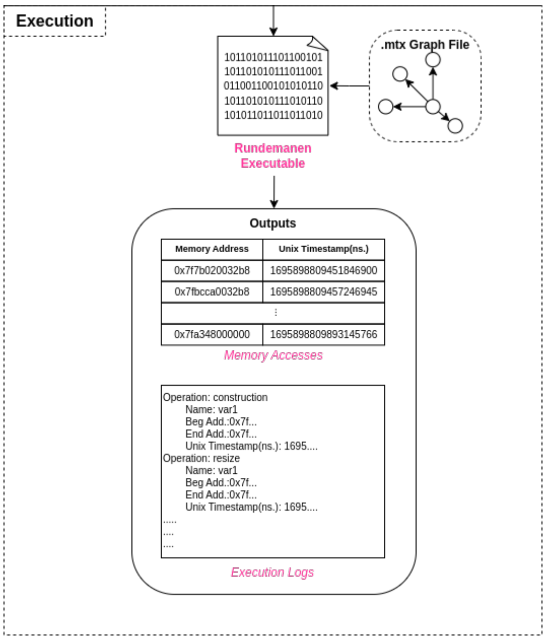
3.2.4.2. Page Faults

CUDA Unified Memory, by default, uses on-demand paging, where if the requested page is not found (on the requested location, either CPU or GPU), a page fault exception occurs, and the OS handles it by relocating the page data to the accessed location. When the memory space is insufficient, a page is replaced based on the eviction policy. In the case of oversubscription, where the complete memory space is allocated, and when the eviction policy does not match the application's access patterns, we have memory thrashing, which causes page movements back and forth. Therefore, identifying object groups with high page fault rates and the pages leading to thrashing and eliminating them may result in performance gain.

To collect page fault counts for each object group, we utilize NVIDIA Nsight



(a) Compilation phase.



Memory Address	Unix Timestamp(ns.)
0x7f7b020032b8	1695898809451846900
0x7fbcca0032b8	1695898809457246945
⋮	
0x7fa348000000	1695898809893145766

```
Operation: construction
        Name: var1
        Beg Add.:0x7f...
        End Add.:0x7f...
        Unix Timestamp(ns.): 1695....
Operation: resize
        Name: var1
        Beg Add.:0x7f...
        End Add.:0x7f...
        Unix Timestamp(ns.): 1695....
.....
....
....
```

(b) Execution phase.

Figure 3.4. The steps of the memory accesses' data collection.

Systems to monitor the addresses of pages experiencing faults. Profiling of applications is carried out by utilizing the Nsight System's command-line tool with data exported to a *.sqlite* file. A dedicated Python script has been developed to extract information pertaining to page faults. This script parses the *.sqlite* file and execution logs, associating each faulted page address with a specific object. It's important to note that only entries within the memory copy table are considered, with a specific focus on page faults occurring between host and device memory.

3.2.5. Artificial Oversubscription Scenerios

To evaluate the system performance under different oversubscription rates, we configure a set of oversubscription scenarios, where we pre-allocate memory space on the GPU device memory using *cudaMalloc* as it is done in (Shao et al. 2022). Specifically, we generate five different oversubscription scenarios in which varying percentages of the required space of the graph are pre-allocated on the global memory. The scenarios are No, 10%, 30%, 50%, and 70% oversubscriptions. To calculate how much memory space we need to pre-allocate, we use the following formula:

$$(Mem.Size) - (1 - \frac{Oversubscription\%}{100})x(Mem.Req.)$$

where $Mem.Size$ is the total GPU global memory size and $Mem.Req.$ is the maximum (peak) memory requirement of the application.

3.2.6. Data-Access Aware UM Advises

This thesis investigates how migration policies applied at the data structure level impact performance using various UM Advises. Table 4.5. shows which advice is used for each object group. The advises are created considering the memory accesses and page faults shown in Preliminary Experiment (Section 4.4.).

For each object, a memory advice is applied right after construction and resizing using the *cudaMemAdvise* function. If it is required to set both the *prefLoc.* and *accBy.* for the object, the function is called twice: one with *cudaMemAdviseSetPreferredLocation* and the other with *cudaMemAdviseSetAccessedBy*. Listing 3.4. shows how these are applied to *indices* of *Graph's CSR*.

Listing 3.4. An example C++ that applies memory advice to a thrust object.

```
inline int get_device_id() {
    int device_id;
    cudaGetDevice(&device_id);
    return device_id;
}
#define THRUST_RAW_PTR_CAST(x) thrust::raw_pointer_cast(x.data())
#define THRUST_PTR_TYPE(x) std::remove_pointer<decltype(THRUST_RAW_PTR_CAST(x))>::type

... function(...) {
    #ifdef UM_OTHERS_ADVISE1_ENABLED
    cudaMemAdvise(THRUST_RAW_PTR_CAST(indices),
                  sizeof(THRUST_PTR_TYPE(indices)) *  indices.size(),
                  cudaMemAdviseSetPreferredLocation,
                  cudaCpuDeviceId);
    #endif
    #ifdef UM_OTHERS_ADVISE2_ENABLED
    cudaMemAdvise(THRUST_RAW_PTR_CAST(indices),
                  sizeof(THRUST_PTR_TYPE(indices)) *  indices.size(),
                  cudaMemAdviseSetAccessedBy,
                  get_device_id());
    #endif
}
```

CHAPTER 4

EXPERIMENTAL STUDY

In this chapter, we introduce the datasets used throughout this study, provide meta information about them, collect and analyze memory accesses and page faults for each, and, by considering the results, suggest which memory advice hint to apply. The effects of memory advices are discussed in the next chapter.

4.1. Machine Specification and Compilation

The application was compiled using GCC 10.5.0 and CUDA 12.0 Toolkit. Table 4.1. displays the testing environments and their specifications. We selected two machines, each featuring a distinct GPGPU microarchitecture, to demonstrate that the memory advice we will introduce performs well across different microarchitectures. This choice also aims to emphasize any variations resulting from architectural differences, if present.

Table 4.1. The testing environments and their specifications.

Test Environment	**Specification**
PASCAL	2 x Xeon® Silver 4114 CPU 32GB RAM 8GB Quadro P4000 GPU
AMPERE	Xeon® E5-2609 v4 64 GB RAM NVIDIA RTX 3060 Ti 8GB

4.2. Datasets

Table 4.2. shows the datasets used in the experiments. All datasets are loaded in MatrixMarket (.mtx) file format. After loading the graph, any self-loop is deleted.

Table 4.2. Datasets used in the experiments.

Test Env.	**Dataset**	**|V|**	**|E|**	**Symmetric**	**Weighted**	**|E| After Conversion**
PASCAL & AMPERE	audikw_1	943,695	39,297,771	Yes	Yes	76,708,152
	soc-LiveJournal1	4,847,571	68,993,773	No	No	68,475,391
	Long_Coup_dt6	1,470,152	44,279,572	Yes	Yes	85,618,840
	dielFilterV3real	1,102,824	45,204,422	Yes	Yes	88,203,196
	cage15	5,154,859	99,199,551	No	Yes	94,044,692
	rgg_n_2_24_s0	16,777,216	132,557,200	Yes	No	265,114,400
	kron_g500-logn21	2,097,152	91,042,010	Yes	Yes	182,081,864
	uk-2002	18,520,486	298,113,762	No	No	292,243,663
AMPERE	uk-2005[1]	39,459,925	936,364,282	No	No	921,345,078
	kmer_V1r[2]	214,005,017	232,705,452	Yes	No	465,410,904
	kron_24_24[3]	16,777,216	386,959,739	Yes	No	773,919,478
	sk-2005[4]	50,636,154	1,949,412,601	No	No	1,930,292,948

Note: For kron_A_B, A is the scale used in the number of vertices the graph would include($|V| = 2^A$), and B represents the average vertex degree.

4.3. Peak Memory Requirements

Table 4.3. displays the peak memory requirements for each dataset's object groups during runtime collected in the PASCAL environment. In general, *Others* consumes the most memory space, followed by *Graph's CSR*, *Community Info.*, and finally, *Hash Tables*.

The higher memory requirement of *Hash Tables* for kron_g500-logn21 can be attributed to the means of the degrees of vertices being high for both the *bucket319toInf* and *bucket84to318* (see Algorithm 2) in the Movement phase and *commBucket480toInf* and *commBucket128to479* (see Algorithm 3) in the Aggregation phase. Additionally, for some datasets, *Community Info* demands more memory due to having relatively small

1. (Boldi and Vigna 2004; Boldi et al. 2011; Boldi et al. 2004; Davis and Hu 2011)
2. (Benson et al. 2012; Davis and Hu 2011)
3. (Jurij Leskovec and Faloutsos 2005; Beamer, Asanovic, and Patterson 2015)
4. (Boldi and Vigna 2004; Boldi et al. 2011; Boldi et al. 2004; Davis and Hu 2011)

average degrees of vertices; in other words, compared to the number of edges, the number of vertices is high. Consequently, the initial number of communities and their associated data are substantially more.

Table 4.3. Object groups and their peak memory requirements in MB for all datasets collected in the PASCAL environment.

Dataset	**Graph's CSR**	**Hash Tables**	**Community Info.**	**Others**
audikw_1	617	43	30	890
soc-LiveJournal1	293	391	155	606
Long_Coup_dt6	690	29	47	1,057
dielFilterV3real	710	30	35	926
cage15	772	99	164	941
rgg_n_2_24_s0	1,127	2	536	2,188
kron_g500-logn21	1,465	775	67	2,022
uk-2002	1,243	314	592	2,451

4.4. Preliminary Analysis Experiments

We conducted preliminary experiments in the PASCAL environment to determine which memory hints to apply to which object group. We generated charts for memory accesses and page faults to identify the pages migrating back and forth between host and device memories, leading to memory thrashing.

4.4.1. Memory Accesses

Table 4.4. shows the numbers of memory accesses and Figure 4.1. depicts memory access ratios for each object group across all datasets, specifically at the 0% oversubscription as memory requests remain consistent across oversubscription scenarios. With the exception of kron_g500-logn21, *Hash Tables* exhibits low memory accesses compared to other object groups. *Graph's CSR* and *Community Info.* hold the 1st and 2nd positions for the highest memory accesses, except for uk-2002 where *Others* account for the majority of memory accesses. The average memory access percentages for *Graph's CSR*, *Hash*

Tables, *Community Info. Others* are 35.1%, 5.2%, 33.4% and 26.3%, respectively.

Table 4.4. The numbers of memory accesses of each object group for all datasets collected in the PASCAL environment.

Dataset	**Graph's CSR**	**Hash Tables**	**Community Info.**	**Others**
audikw_1	677×10^6	68×10^6	514×10^6	499×10^6
soc-LiveJournal1	627×10^6	77×10^6	734×10^6	572×10^6
Long_Coup_dt6	917×10^6	18×10^6	925×10^6	603×10^6
dielFilterV3real	$1,564 \times 10^6$	125×10^6	896×10^6	$1,212 \times 10^6$
cage15	595×10^6	36×10^6	681×10^6	331×10^6
rgg_n_2_24_s0	754×10^6	51×10^6	833×10^6	387×10^6
kron_g500-logn21	99×10^6	73×10^6	89×10^6	61×10^6
uk-2002	$1,015 \times 10^6$	80×10^6	$1,055 \times 10^6$	$1,366 \times 10^6$

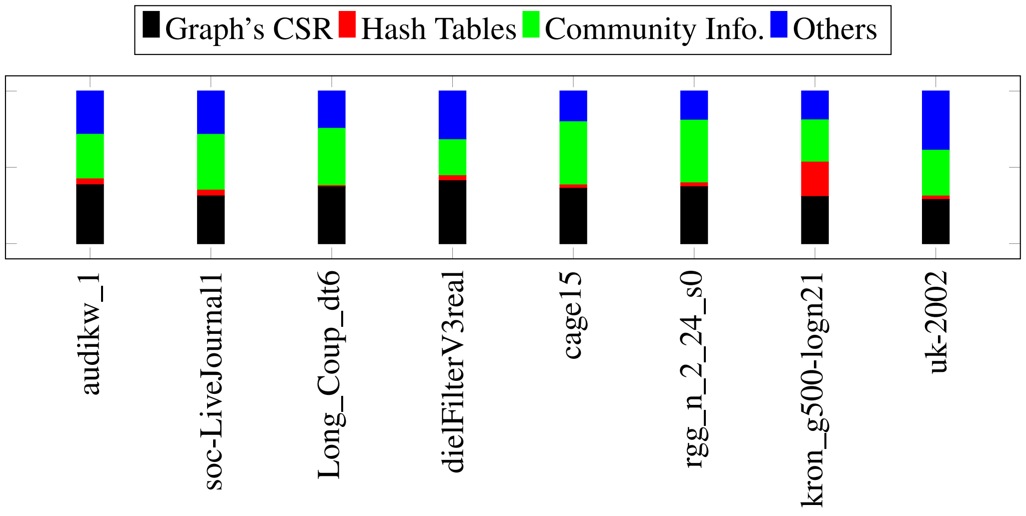
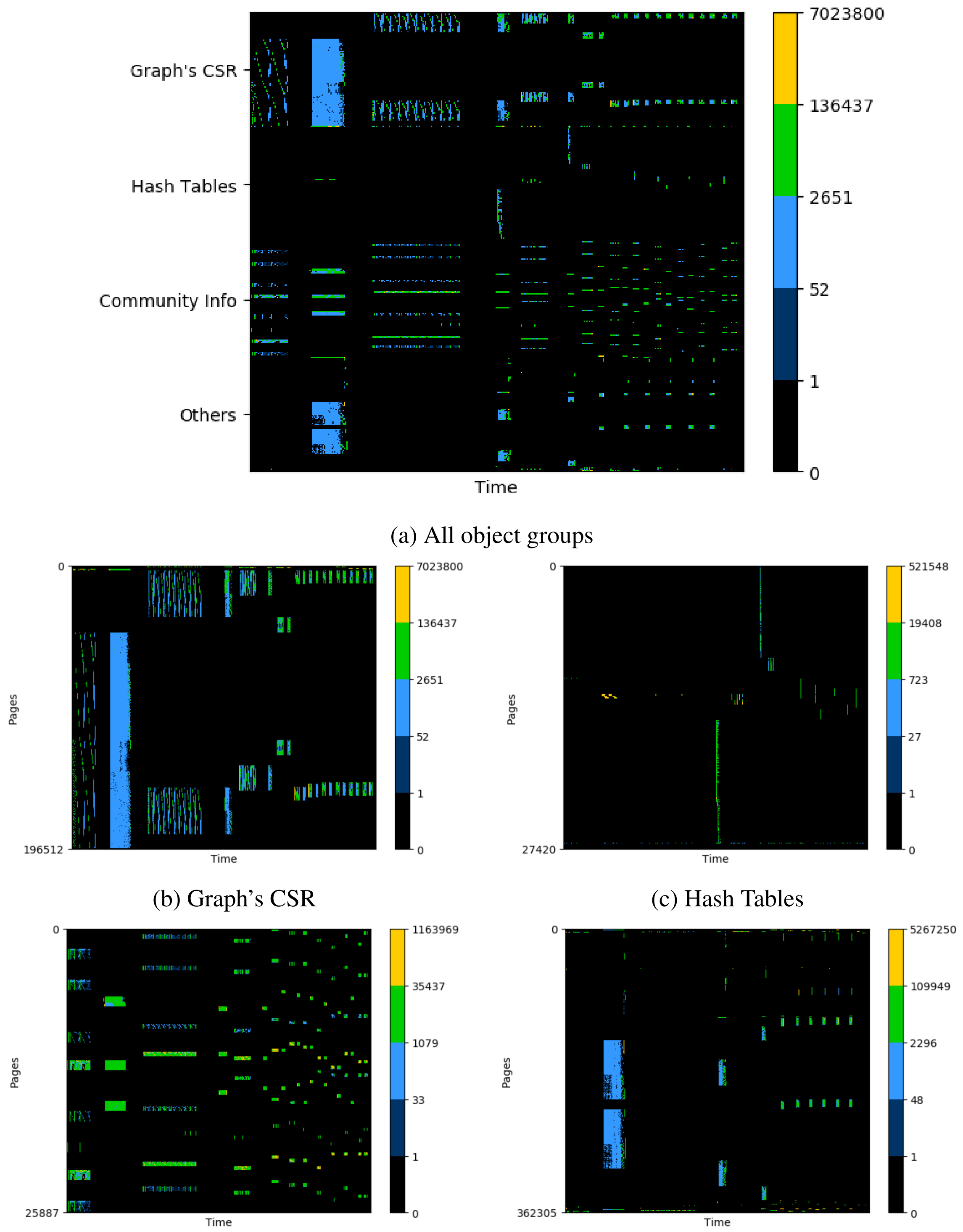


Figure 4.1. Memory access ratios of each object group for all datasets collected in the PASCAL environment.

Figures 4.2. to 4.9. present memory access charts for each object group across all datasets. The y-axis represents pages, and the x-axis represents the time frame of the memory accesses. Each non-black point signifies memory accesses at a specific time frame for a particular page. The color of a point implies the frequency of the memory accesses within the page, with a higher position of a color on the color bar corresponding to more memory accesses at the page, signifying a higher spatial locality —i.e. the addresses of the accesses are very close to each other, for example, they are in the same page. Horizontal non-black points suggest a higher temporal locality, indicating that the same memory addresses are accessed more than once across different time frames. The more horizontally scattered the points, the higher the temporal locality. On the other hand, vertically scattered points imply that there are memory accesses across different pages within the same time frame. The greater the vertical scattering, the higher the memory bandwidth required.

For all datasets, in general, we observe that *Graph's CSR* and *Others* exhibit higher temporal locality due to their densely concentrated areas horizontally. Meanwhile, *Community Info.* demonstrates better spatial locality, indicated by the points on the charts having colors located on top of the color bar (specifically more yellow and green colors). On the other hand, *Hash Tables* has very sparse accesses except for the kron_g500-logn21 dataset, as only this dataset has a higher percentage of memory accesses on *Hash Tables*.



(a) All object groups

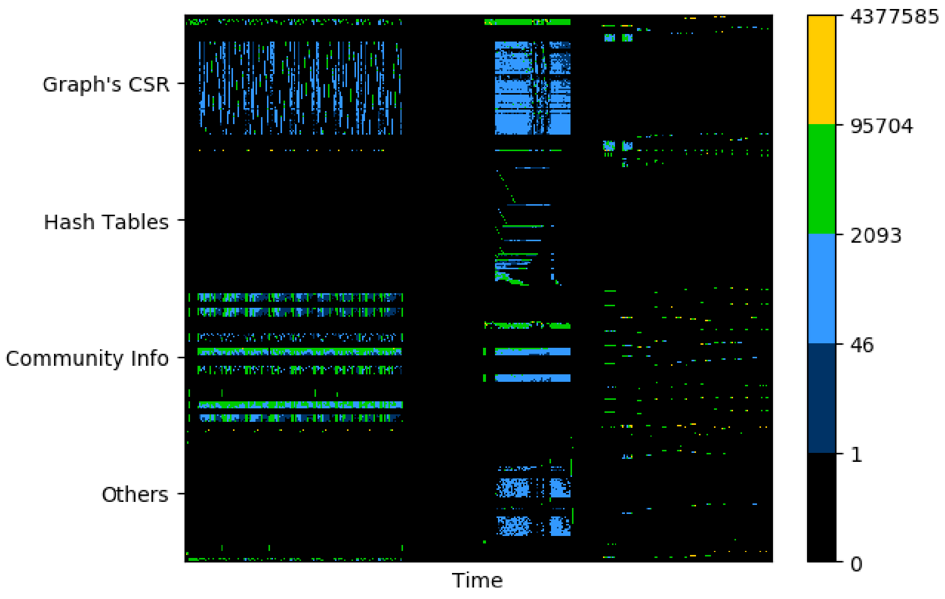
(b) Graph's CSR

(c) Hash Tables

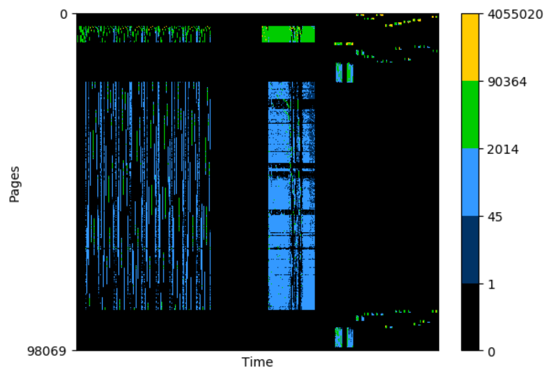
(d) Community Info.

(e) Others

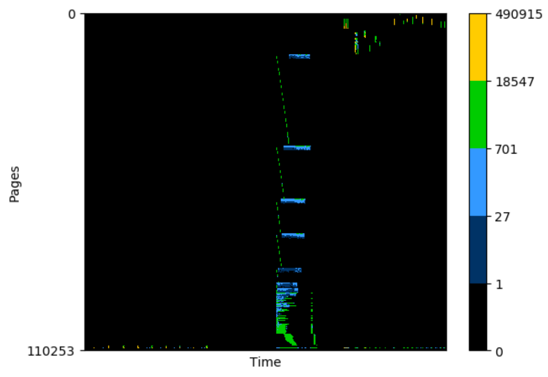
Figure 4.2. Memory accesses of each object group for audikw_1 dataset.



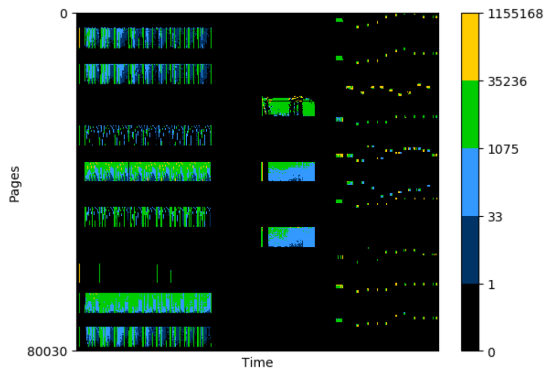
(a) All object groups



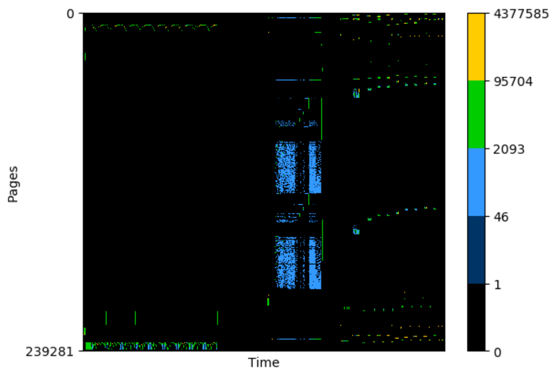
(b) Graph's CSR



(c) Hash Tables

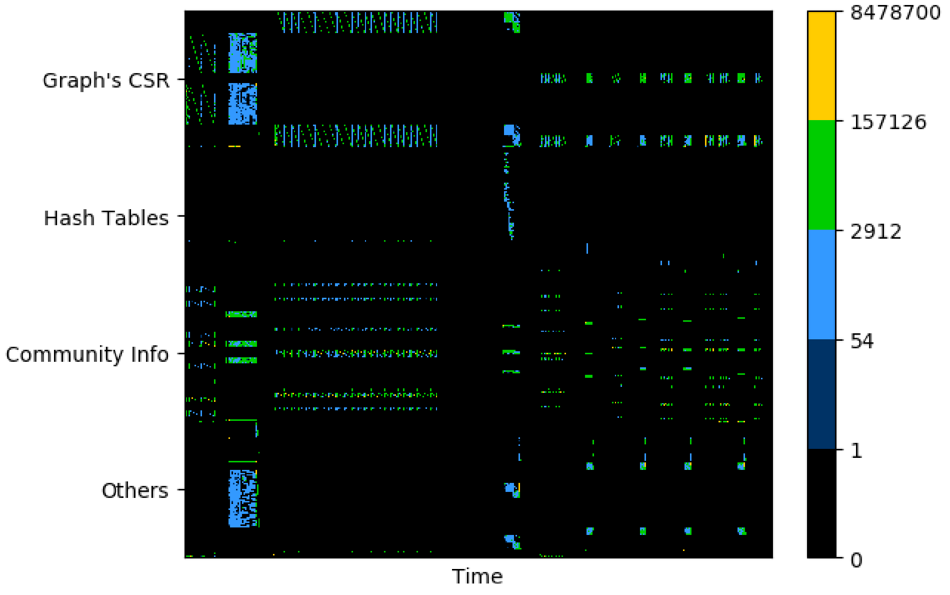


(d) Community Info.

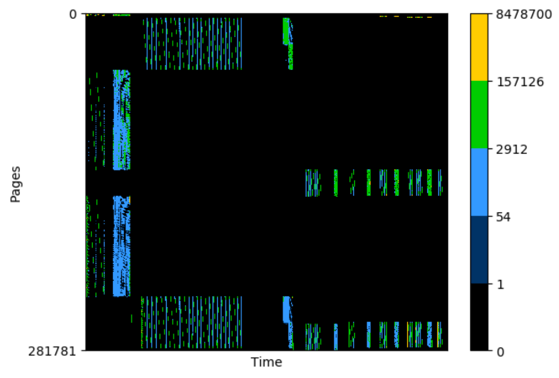


(e) Others

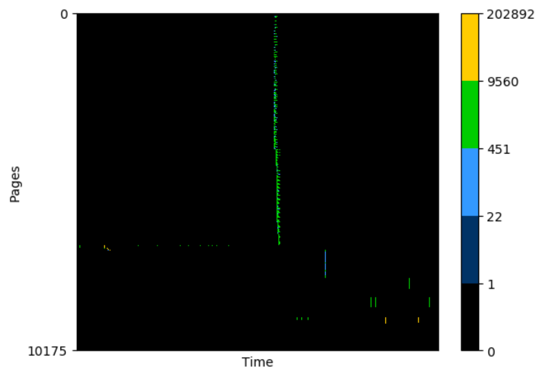
Figure 4.3. Memory accesses of each object group for soc-LiveJournal1 dataset.



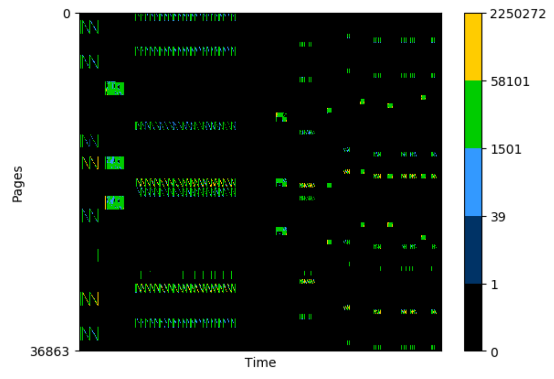
(a) All object groups



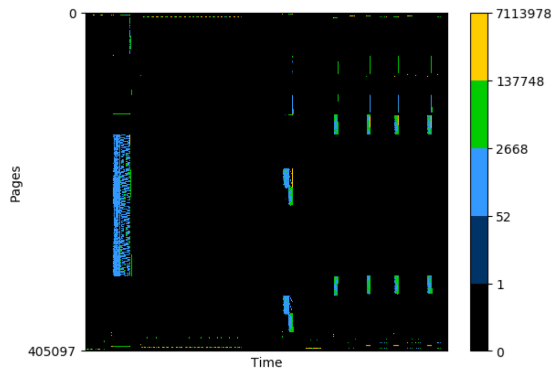
(b) Graph's CSR



(c) Hash Tables

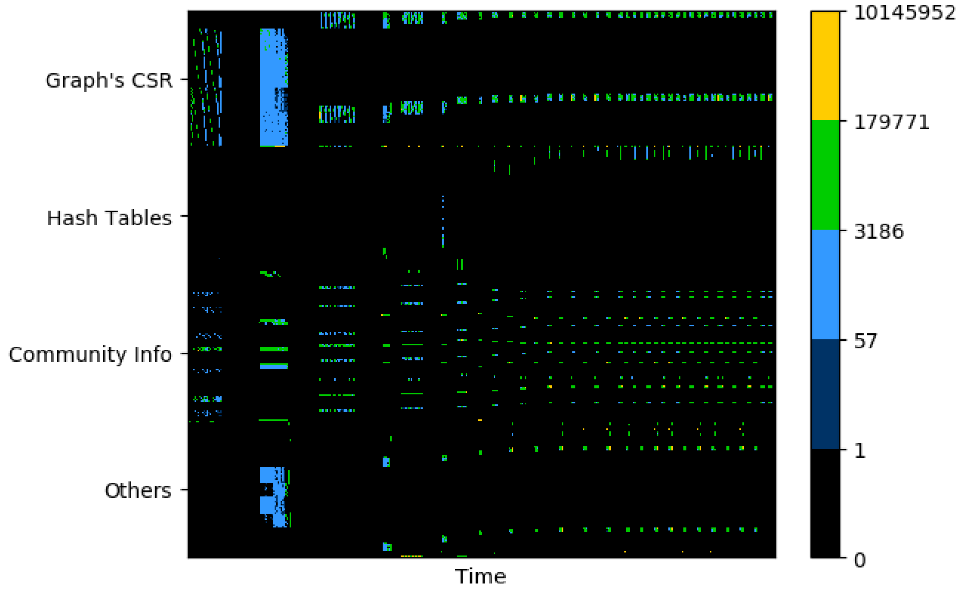


(d) Community Info.

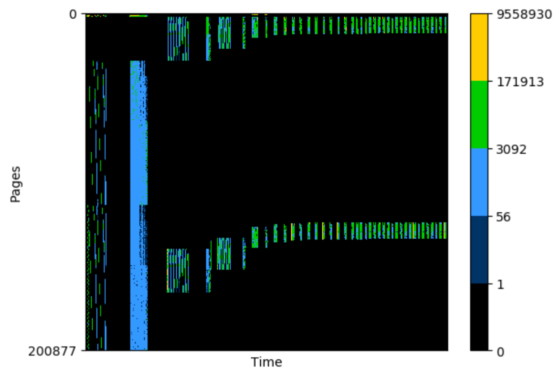


(e) Others

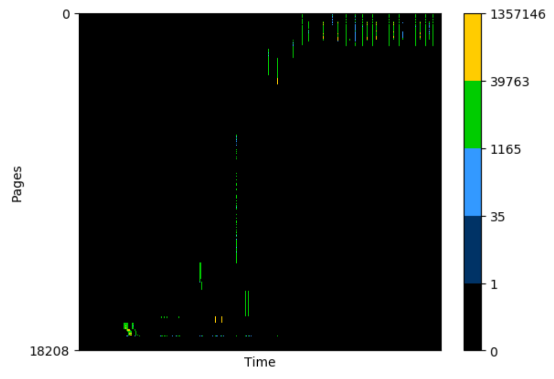
Figure 4.4. Memory accesses of each object group for Long_Coup_dt6 dataset.



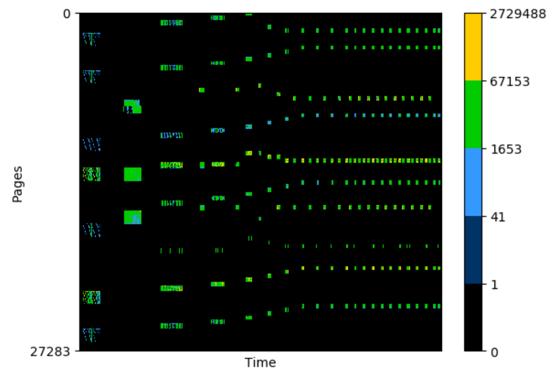
(a) All object groups



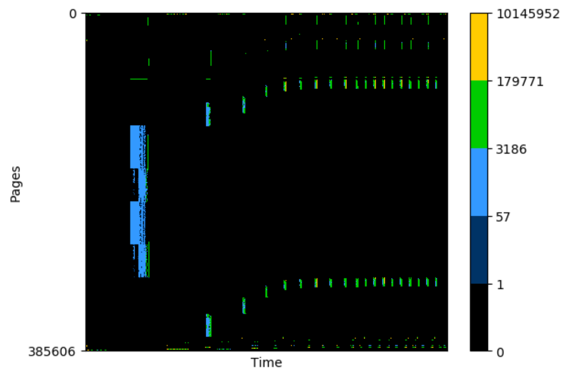
(b) Graph's CSR



(c) Hash Tables

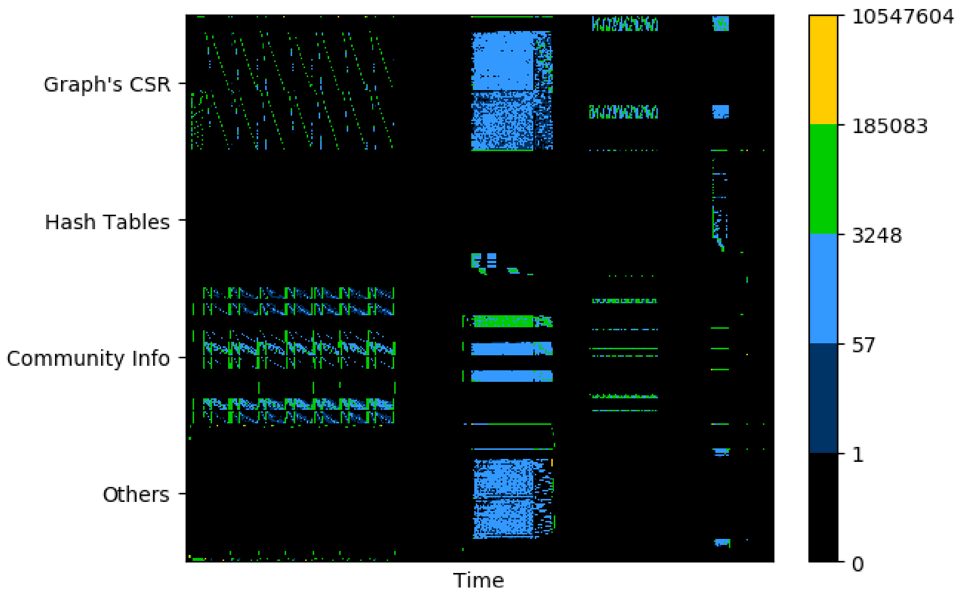


(d) Community Info.

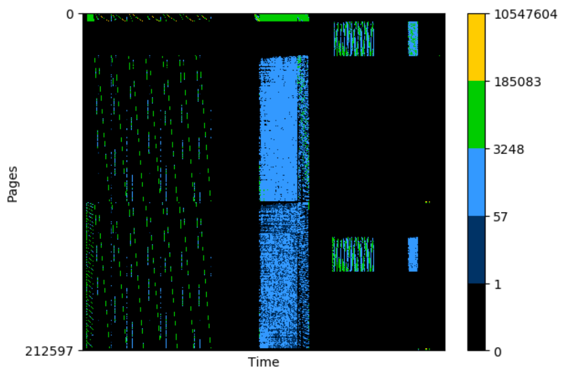


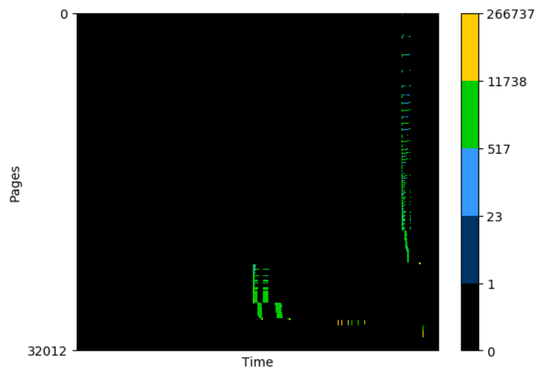
(e) Others

Figure 4.5. Memory accesses of each object group for dielFilterV3real dataset.



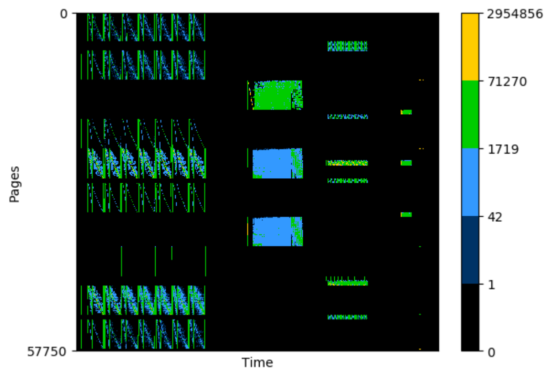
(a) All object groups

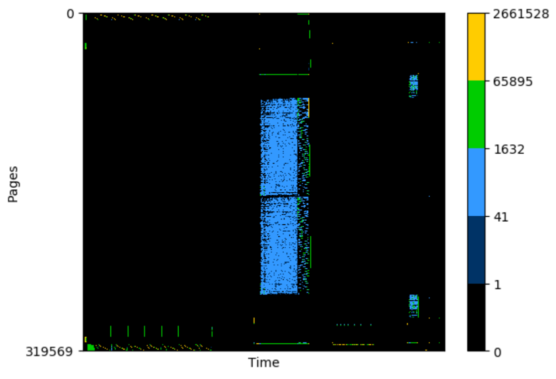




(b) Graph's CSR

(c) Hash Tables

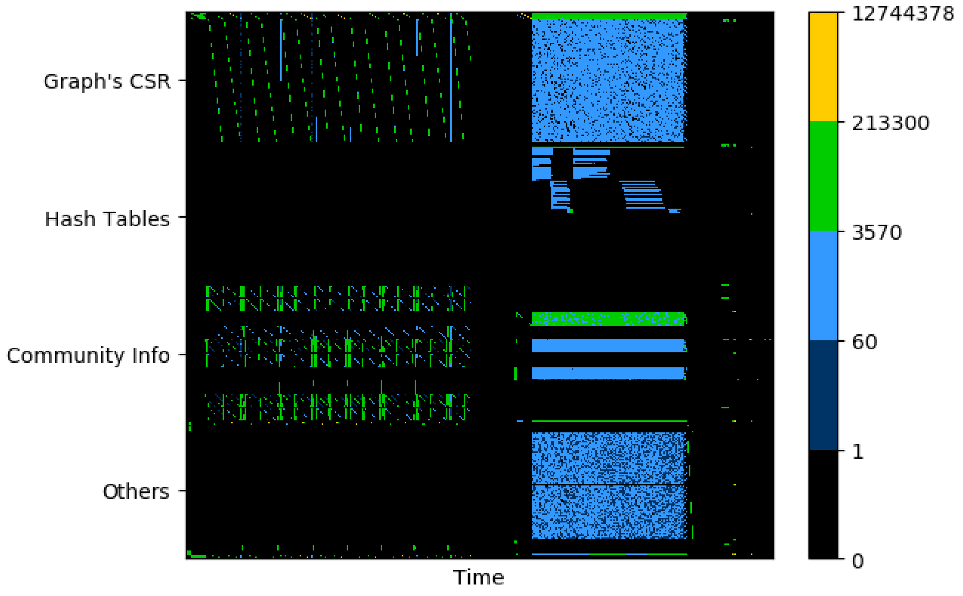




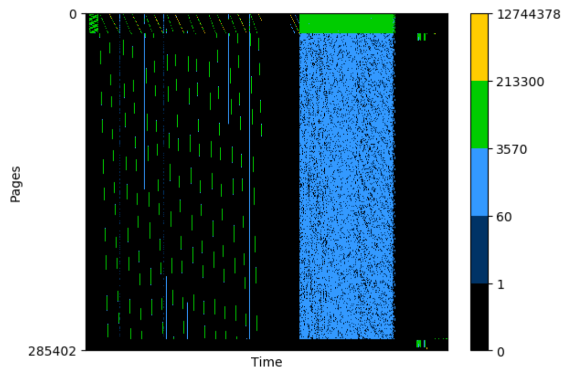
(d) Community Info.

(e) Others

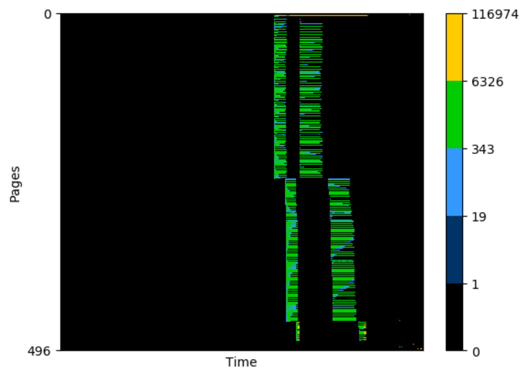
Figure 4.6. Memory accesses of each object group for cage15 dataset.



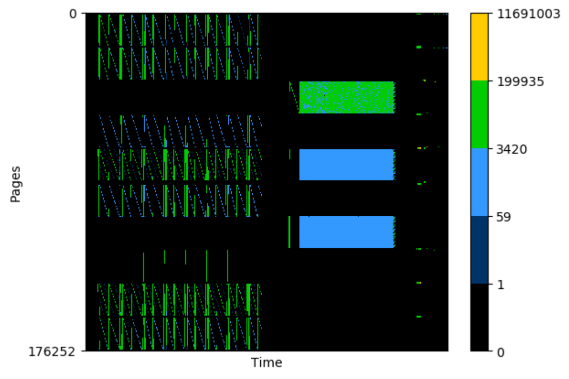
(a) All object groups



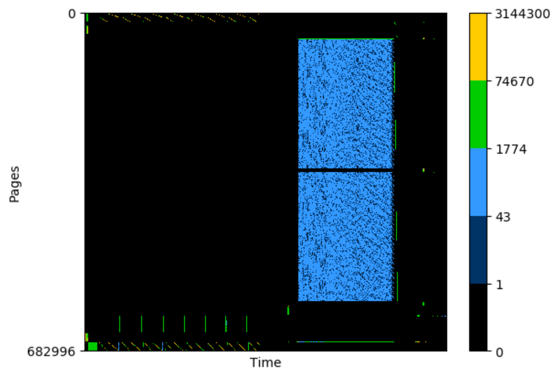
(b) Graph's CSR



(c) Hash Tables

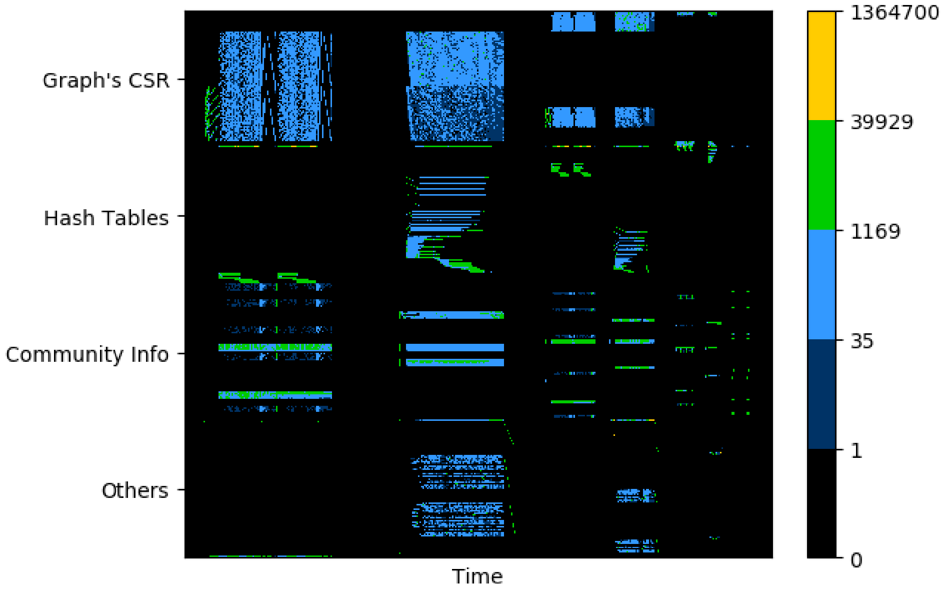


(d) Community Info.

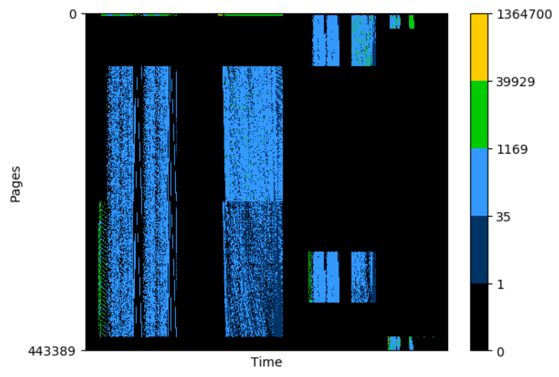


(e) Others

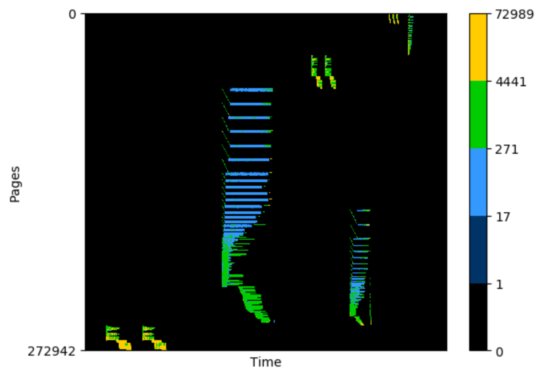
Figure 4.7. Memory accesses of each object group for rgg_n_2_24_s0 dataset.



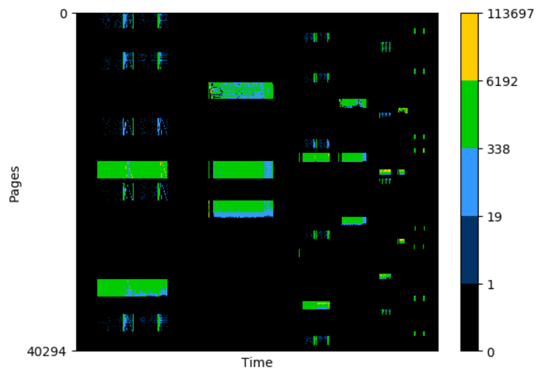
(a) All object groups



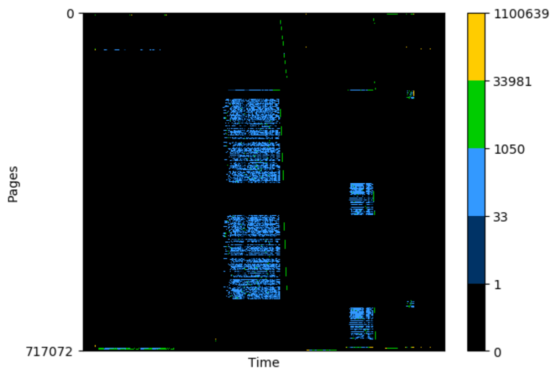
(b) Graph's CSR



(c) Hash Tables

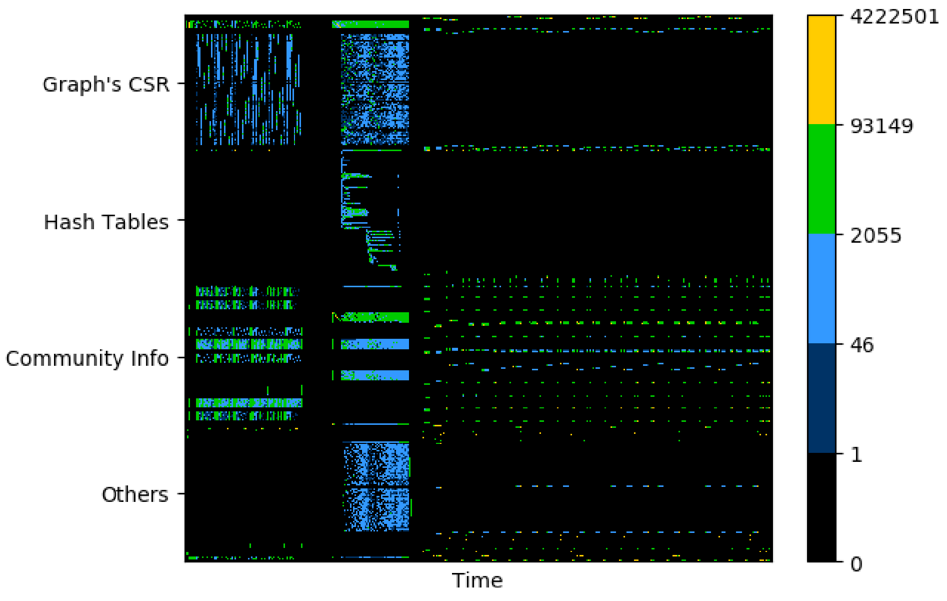


(d) Community Info.

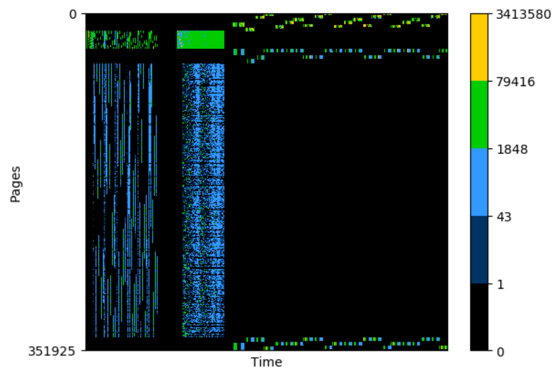


(e) Others

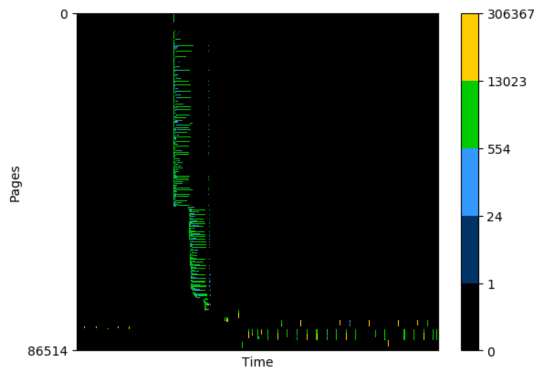
Figure 4.8. Memory accesses of each object group for kron_g500-logn21 dataset.



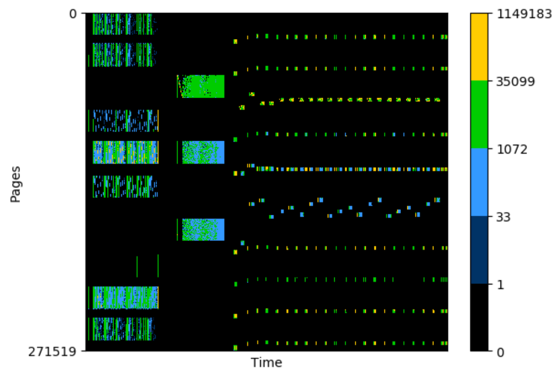
(a) All object groups



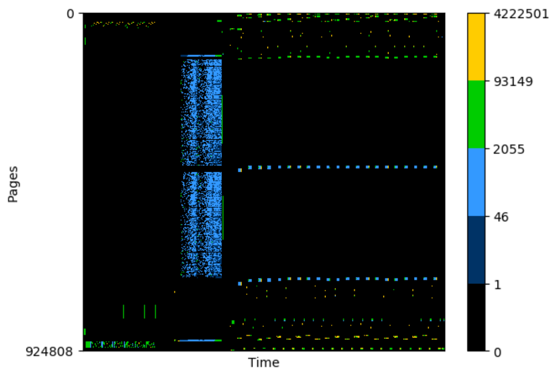
(b) Graph's CSR



(c) Hash Tables



(d) Community Info.



(e) Others

Figure 4.9. Memory accesses of each object group for uk-2002 dataset.

4.4.2. Page Faults

Figure 4.10. presents the page fault ratios for each object group across all datasets at a 30% oversubscription. The data on the chart do not exhibit a one-to-one correspondence with the data in Figure 4.1.. *Graph's CSR* exhibits the highest page fault percentage, and we observe that this percentage is higher for all datasets compared to its percentage of memory accesses for the same datasets. The exception is uk-2002, where the majority of page faults are caused by *Others*. The average page fault percentages for *Graph's CSR*, *Hash Tables*, *Community Info.*, and *Others* are 53.3%, 3.3%, 14.4%, and 29%, respectively.

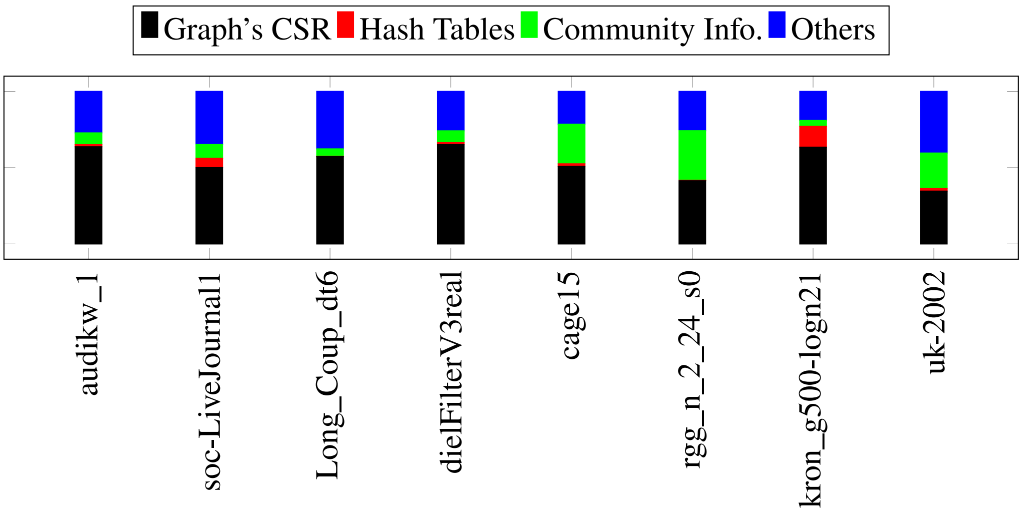


Figure 4.10. Page fault ratios of each object group for all datasets at 30% oversubscription collected in the PASCAL environment.

Figures 4.12. to 4.19. illustrate page fault charts for object groups at a 30% oversubscription rate across all datasets. Similar to the memory access charts, the y-axis represents pages, and the x-axis represents the time of the faults. Each non-black point signifies faults at a specific time frame for a particular page. Horizontal non-black points suggest higher temporal locality, indicating that they are accessed more than once across time frames. The more horizontally scattered the points, the higher the temporal locality. On the other hand, vertically scattered points imply that there are faults of different pages within the same time frame. The greater the vertical scattering, the higher the memory bandwidth required between the host and the device.

As the oversubscription rate increases, the application primarily spends a significant portion of its time in the Aggregate phase during the first iteration, as depicted in Figure 4.11.. The rectangular dense areas in the memory access charts correspond to this phase, and these areas have higher temporal and spatial locality. Therefore, at higher oversubscriptions, this leads to more page faults due to thrashing, resulting in performance drawbacks. To provide clarity, consider the case of a 30% oversubscription, where a substantial amount of time is spent in this phase across all datasets. It's important to note that these noticeable patterns are not reflected in the memory access charts. Consequently, the memory access charts and their corresponding page fault charts at the 30% oversubscription rate do not exhibit similarity because the execution during which the memory access charts are generated did not encounter the page faults that would lead to an increase in running time. Considering the x-axes as unit lengths, for example, the page faults covering the time range from 0.05 to 0.95 in Figure 4.12.(a) correspond to memory accesses in Figure 4.2.(a), spanning the time range between 0.15 and 0.20.

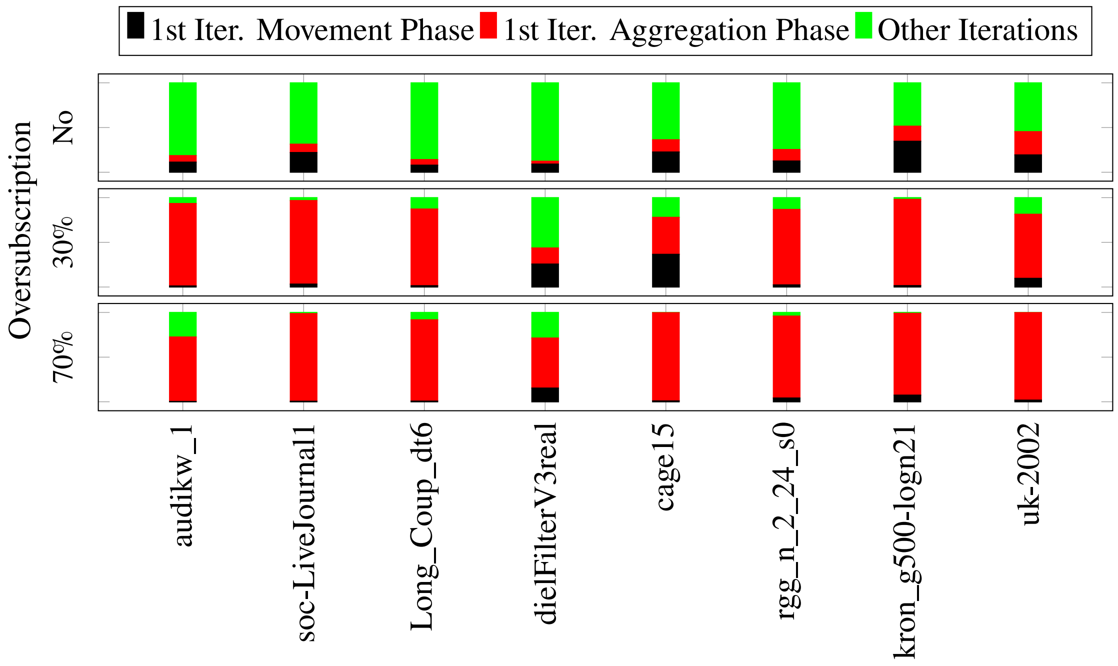
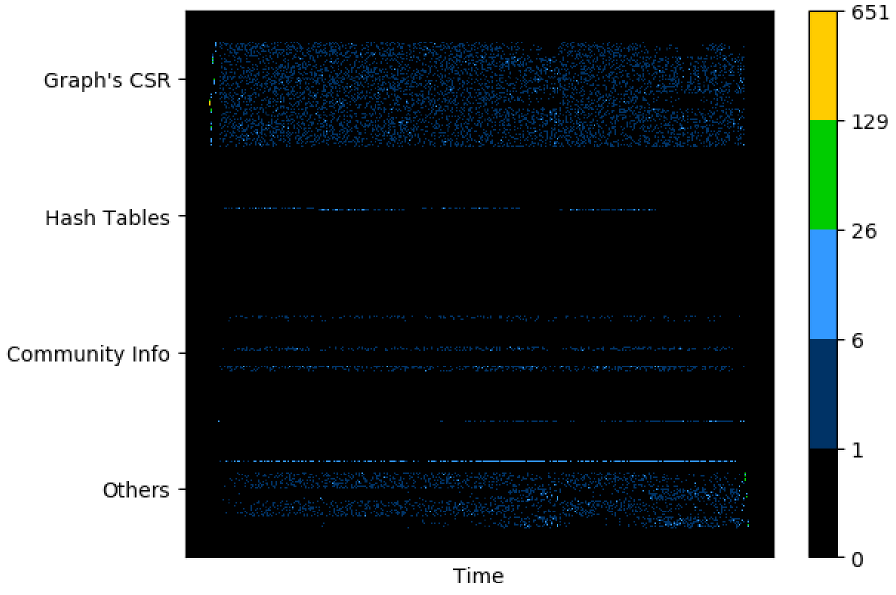
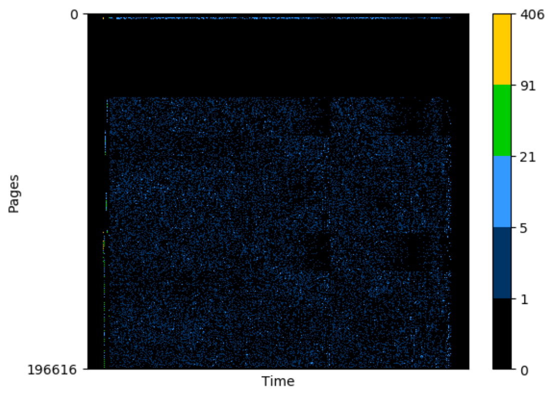


Figure 4.11. Iterations' running times(secs.) of all datasets at No, 30%, and 70% oversubscription configurations collected in the PASCAL environment.

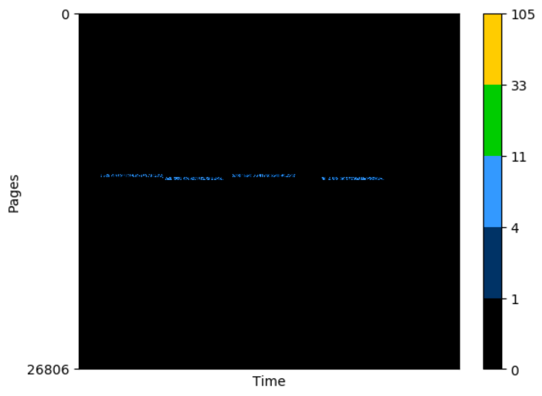
When examining the page fault charts, it is notable that *Graph's CSR* is the primary contributor to the majority of page faults for all datasets, generally followed by *Others*, and occasionally by *Community Info.*. Conversely, *Hash Tables* do not significantly contribute to page faults. Besides, the page faults are densely and horizontally scattered, indicating a significant incidence of memory thrashing.



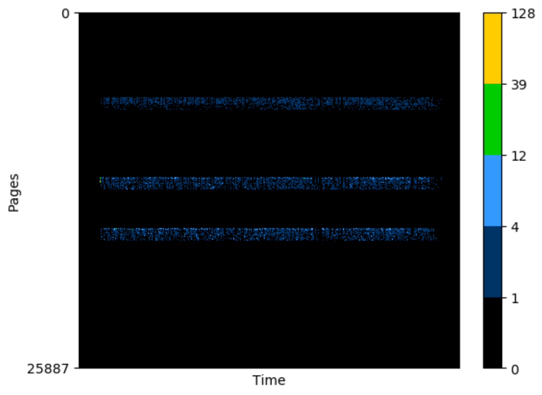
(a) All object groups



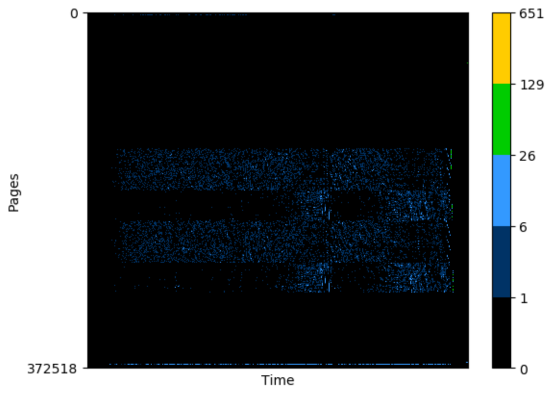
(b) Graph's CSR



(c) Hash Tables

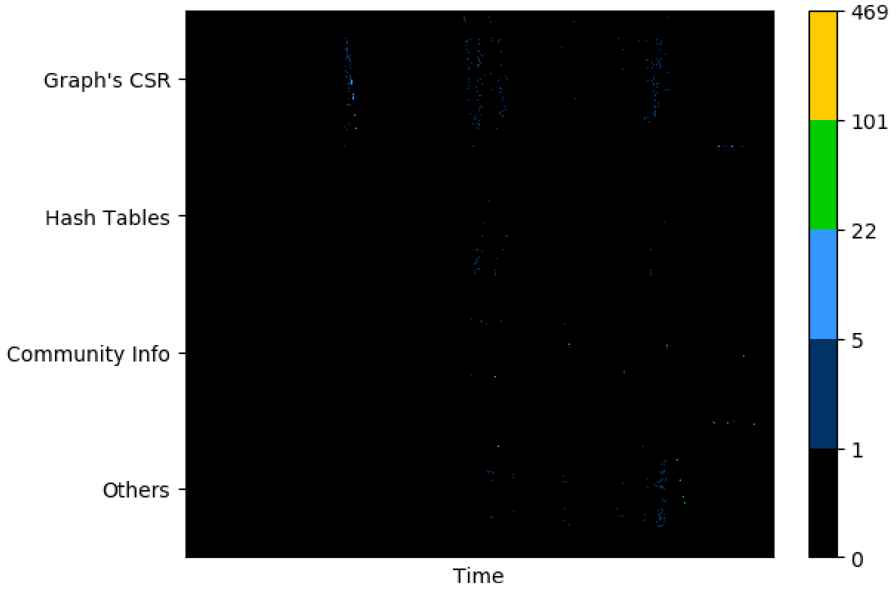


(d) Community Info.

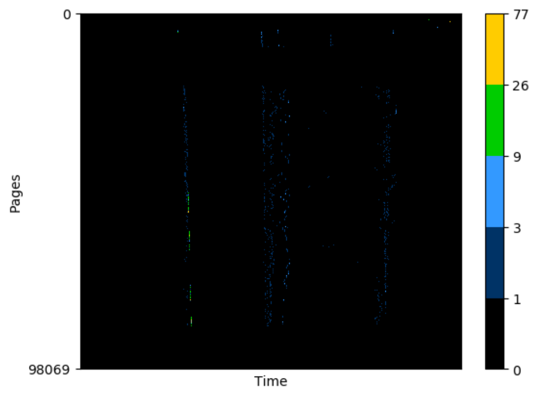


(e) Others

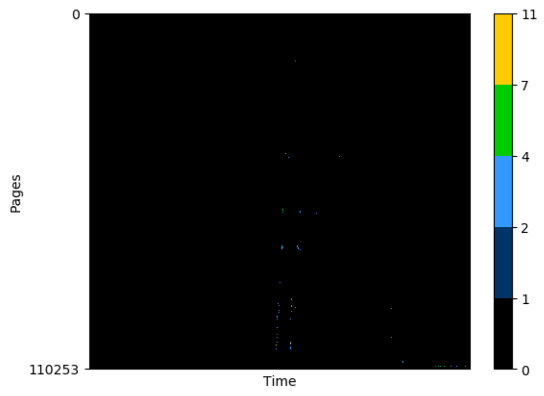
Figure 4.12. Page fault charts of audikw_1 dataset at 30% oversubscription.



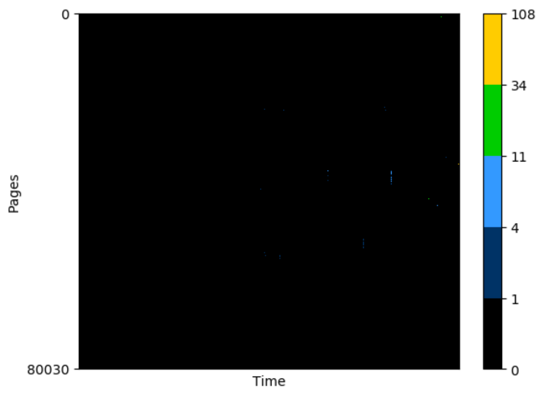
(a) All object groups



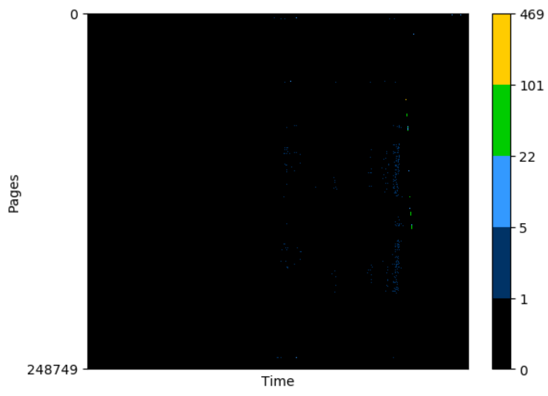
(b) Graph's CSR



(c) Hash Tables

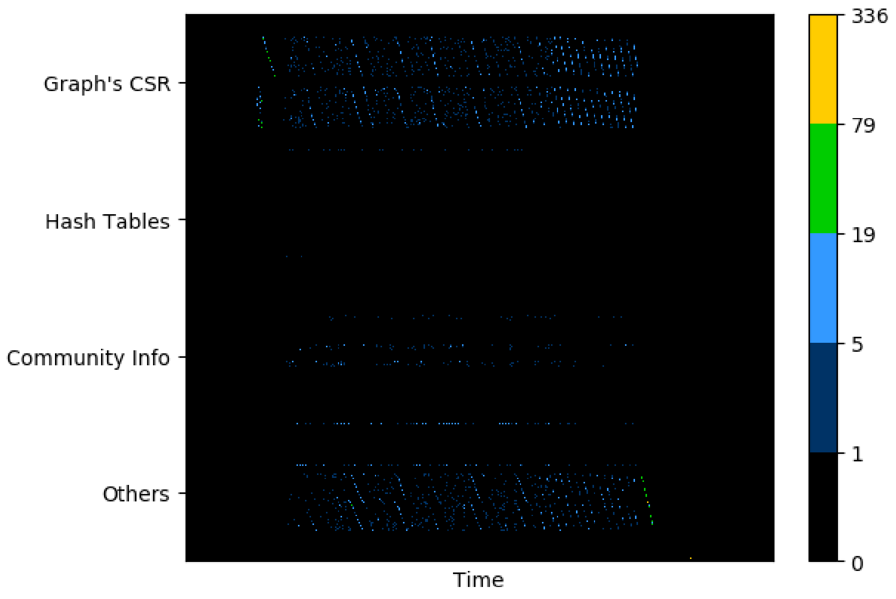


(d) Community Info.

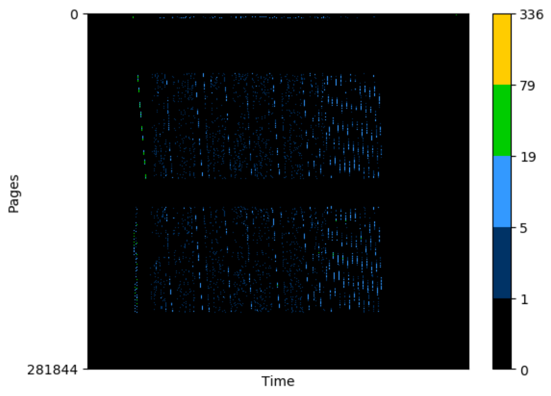


(e) Others

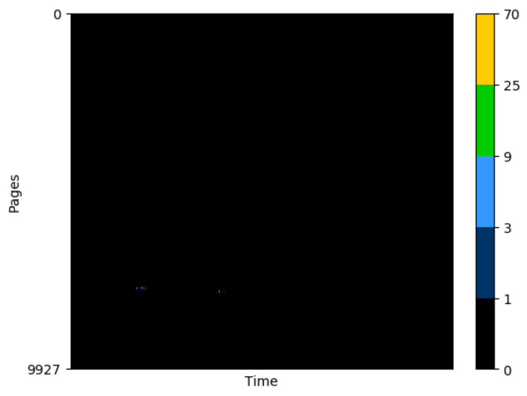
Figure 4.13. Page fault charts of soc-LiveJournal1 dataset at 30% oversubscription.



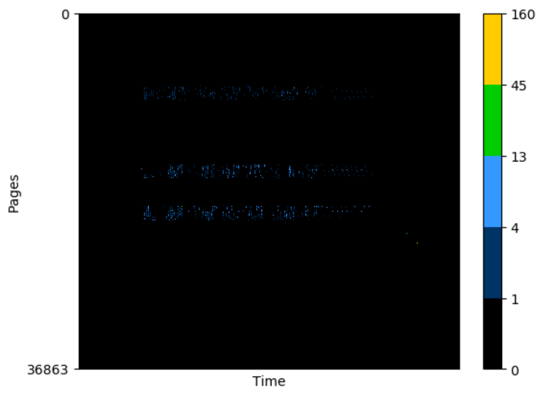
(a) All object groups



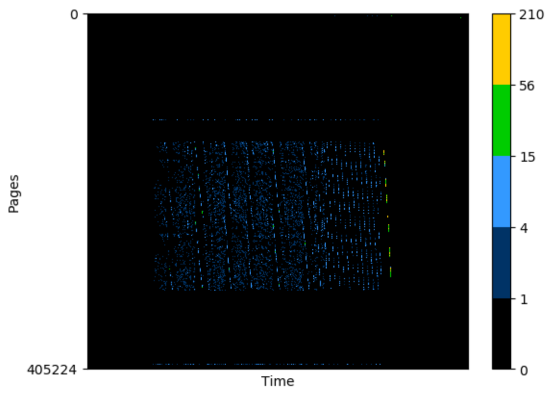
(b) Graph's CSR



(c) Hash Tables

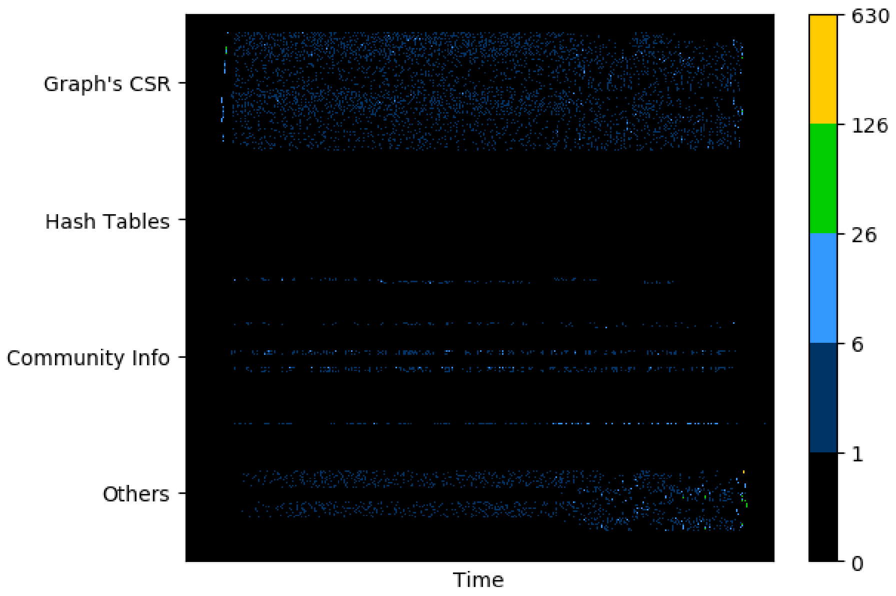


(d) Community Info.

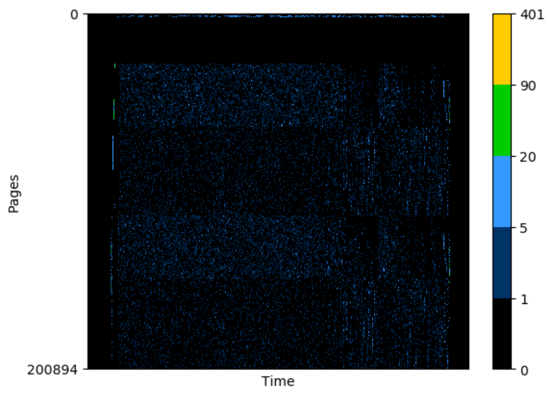


(e) Others

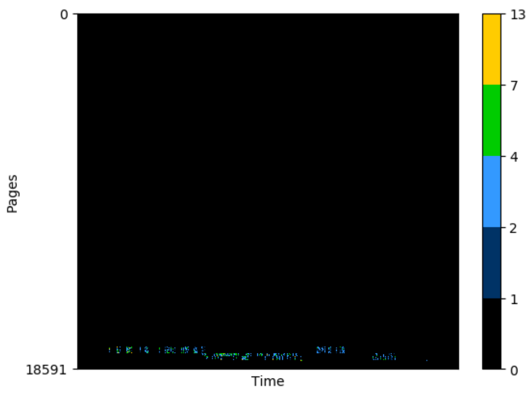
Figure 4.14. Page fault charts of Long_Coup_dt6 dataset at 30% oversubscription.



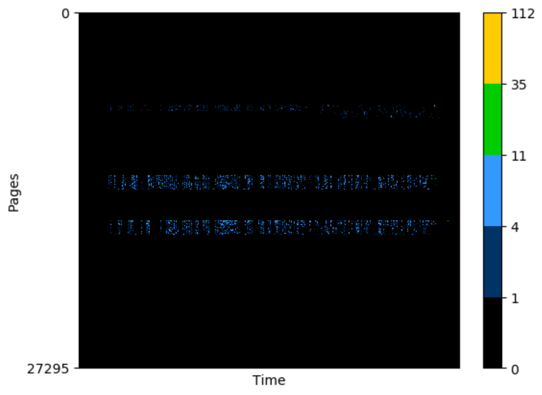
(a) All object groups



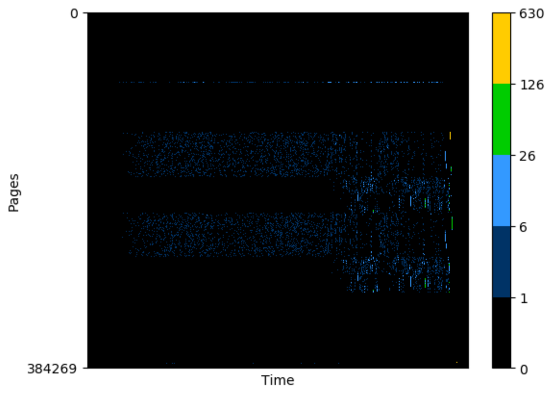
(b) Graph's CSR



(c) Hash Tables

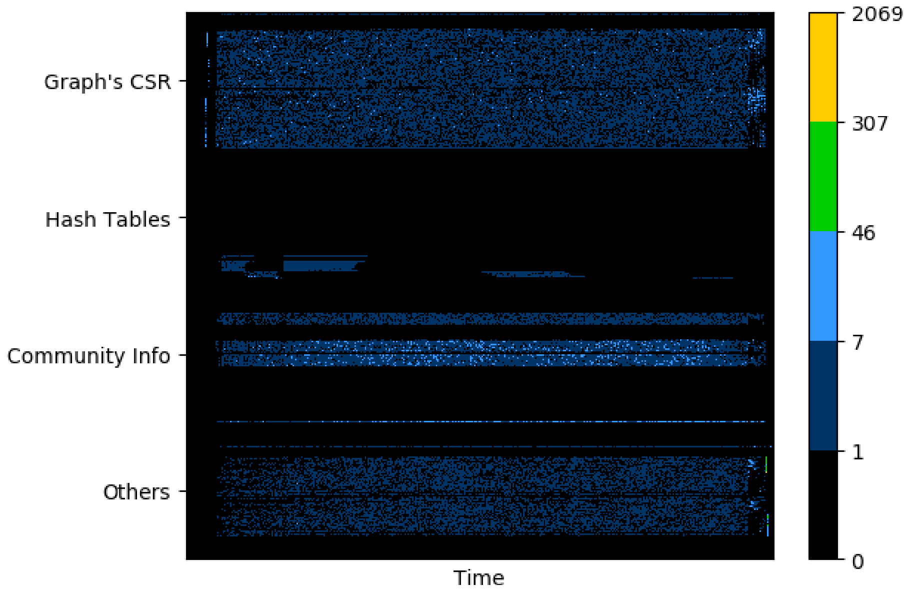


(d) Community Info.

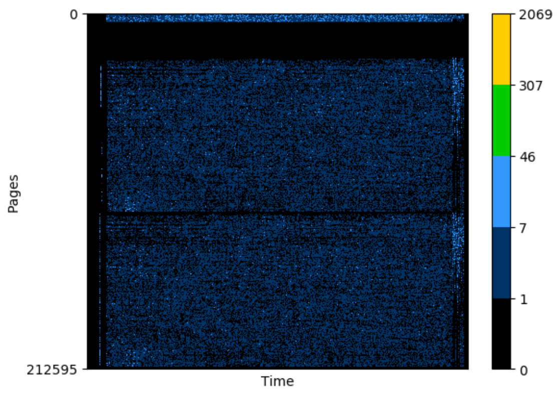


(e) Others

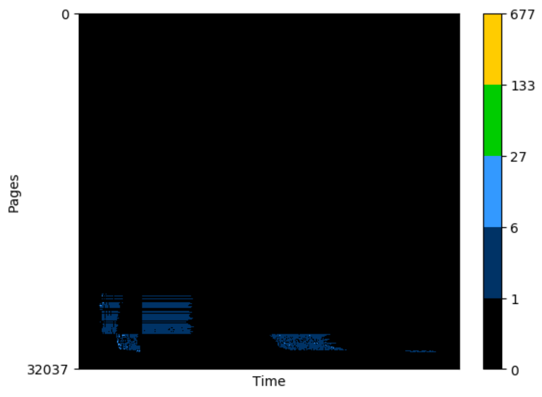
Figure 4.15. Page fault charts of dielFilterV3real dataset at 30% oversubscription.



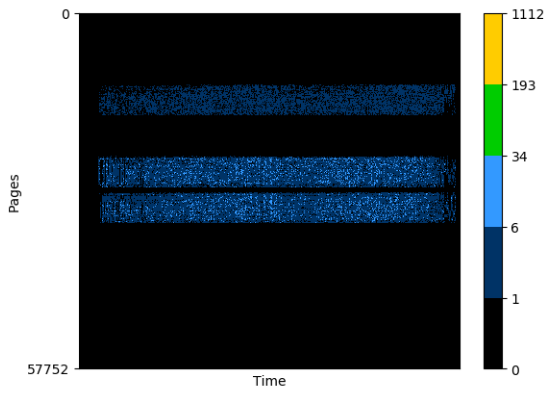
(a) All object groups



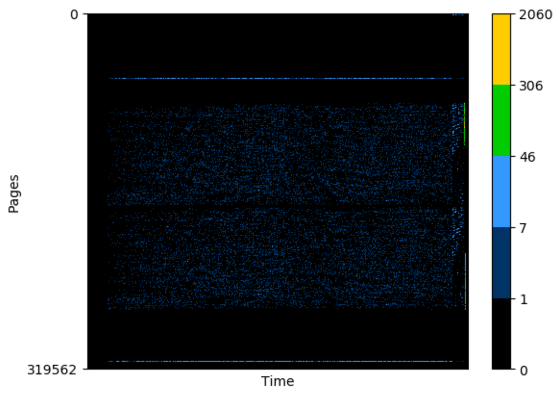
(b) Graph's CSR



(c) Hash Tables

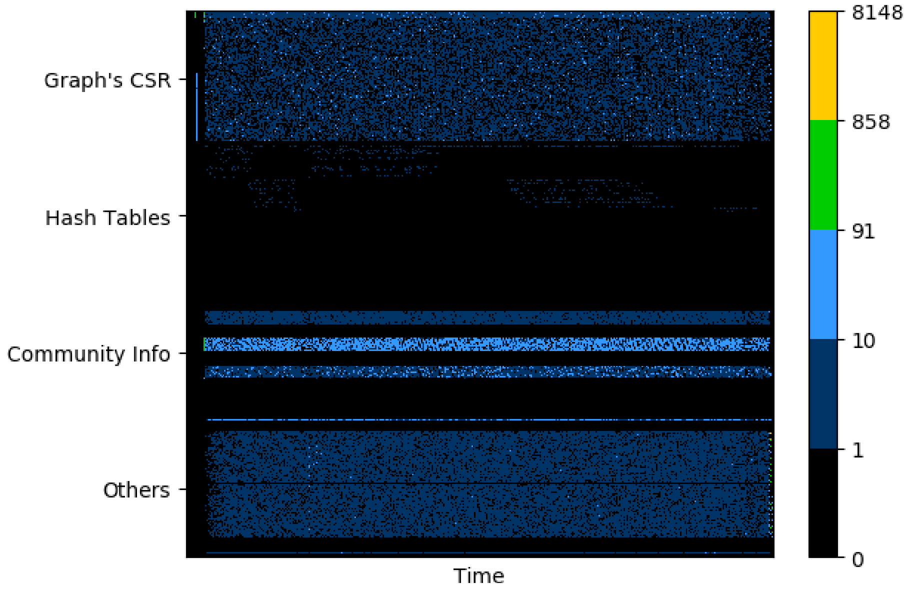


(d) Community Info.

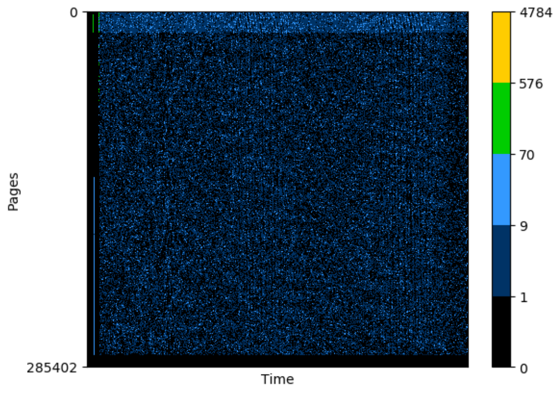


(e) Others

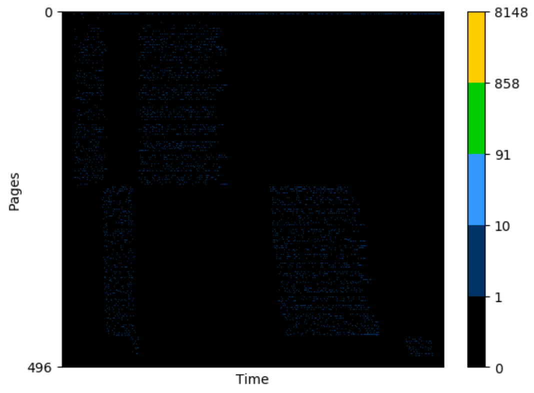
Figure 4.16. Page fault charts of cage15 dataset at 30% oversubscription.



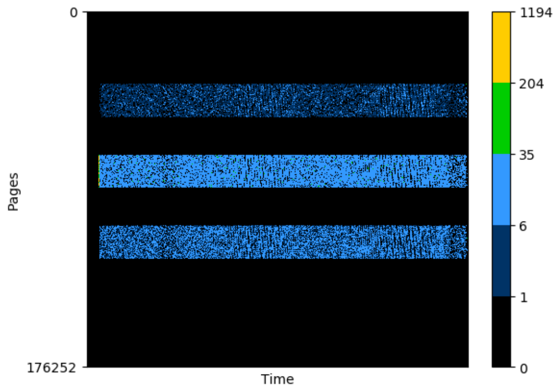
(a) All object groups



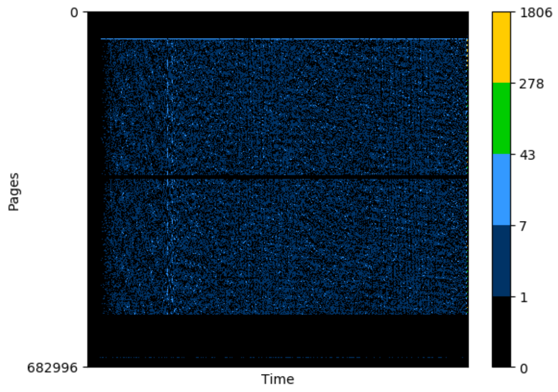
(b) Graph's CSR



(c) Hash Tables

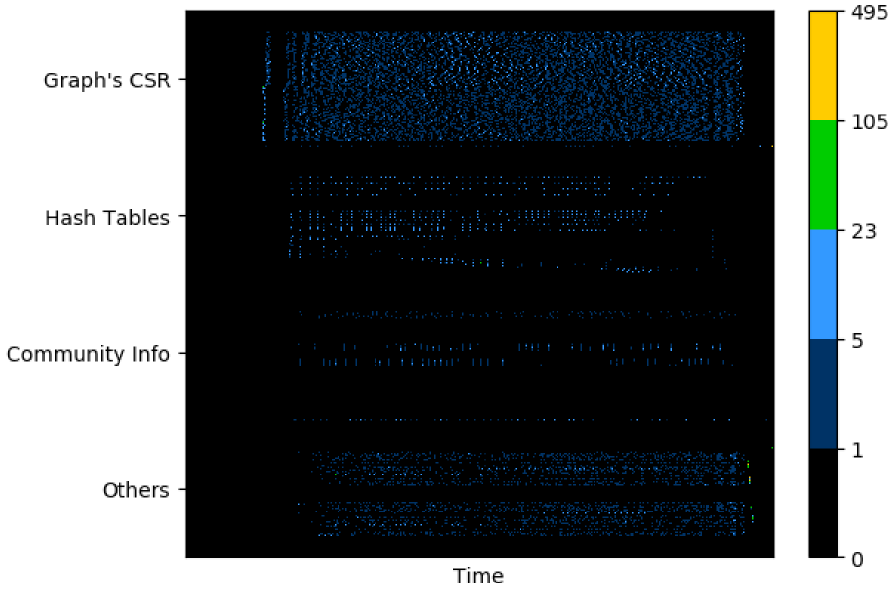


(d) Community Info.

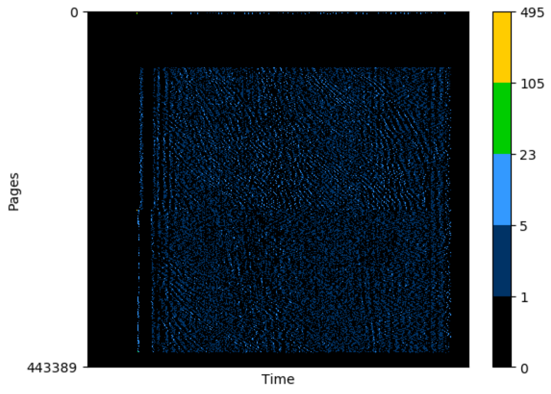


(e) Others

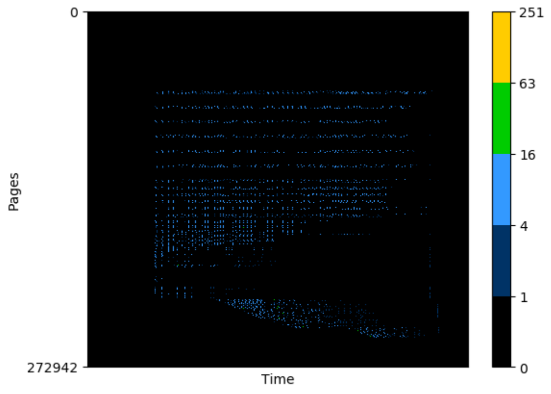
Figure 4.17. Page fault charts of rgg_n_2_24_s0 dataset at 30% oversubscription.



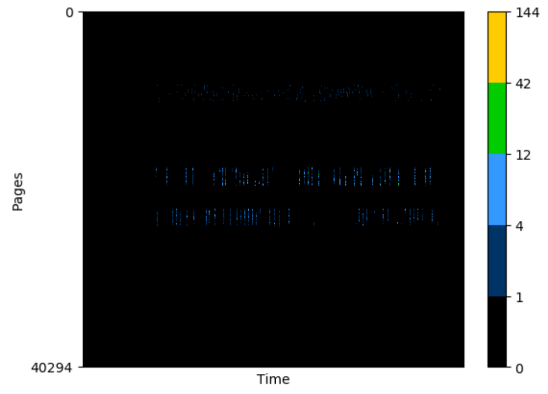
(a) All object groups



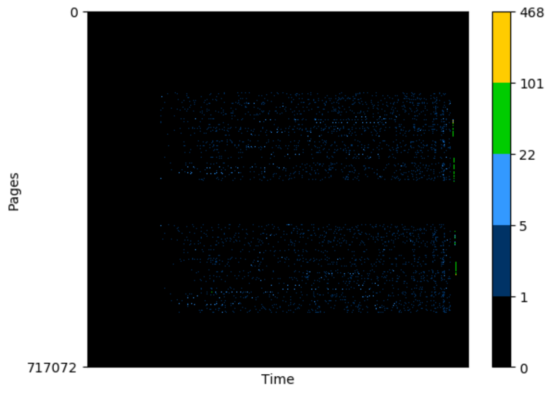
(b) Graph's CSR



(c) Hash Tables

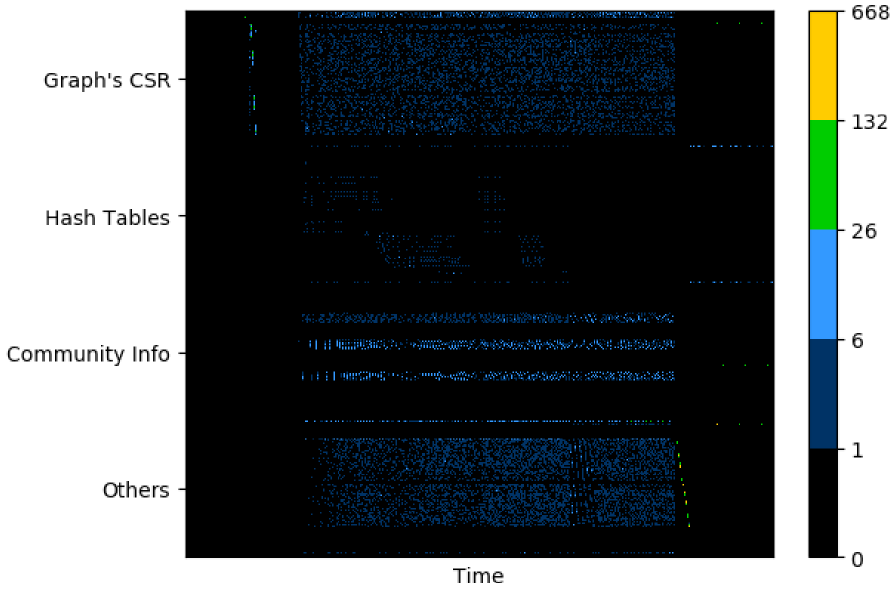


(d) Community Info.

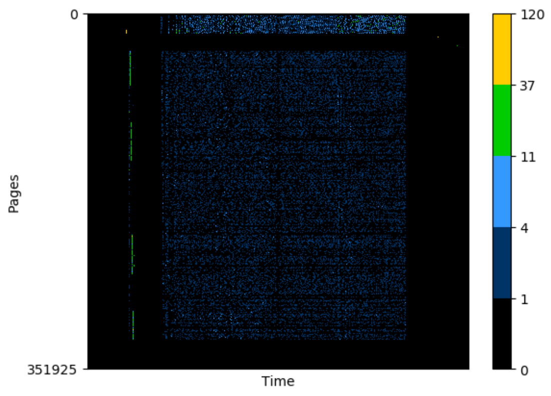


(e) Others

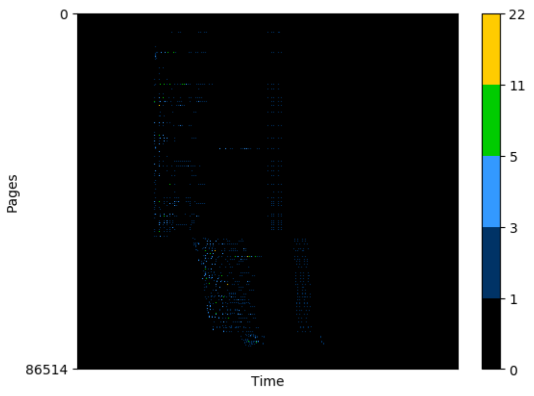
Figure 4.18. Page fault charts of kron_g500-logn21 dataset at 30% oversubscription.



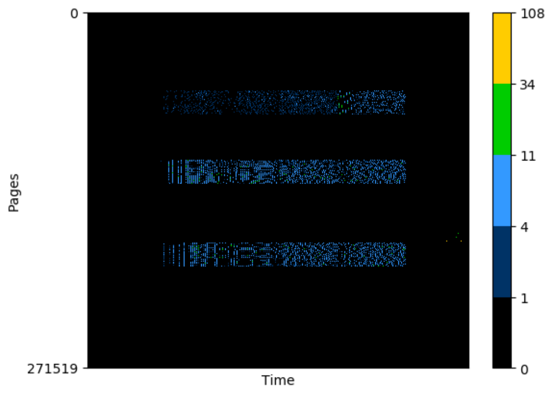
(a) All object groups



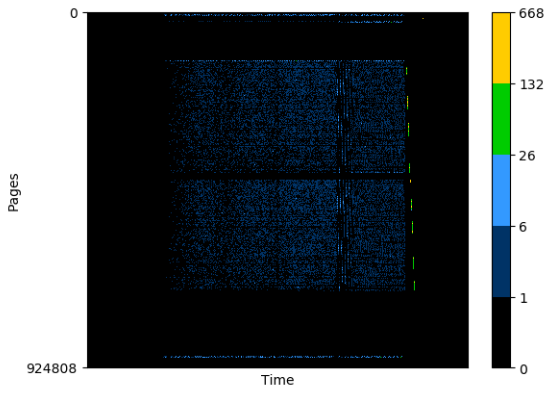
(b) Graph's CSR



(c) Hash Tables



(d) Community Info.



(e) Others

Figure 4.19. Page fault charts of uk-2002 dataset at 30% oversubscription.

4.5. Memory Advises

Considering the Memory Accesses Section (4.4.1.) and the Page Faults Section (4.4.2.), we observe that having a higher spatial locality of memory accesses has no impact on page faults because it implies that the accessed memory addresses are in the same page, and therefore, it does not cause page faults.

Table 4.5. shows the configurations for memory advises applied to object groups. Aside from the naive version, called *base*, we create *adv1* to observe the impact of all object groups together. Then, we define four different advise configurations to test each object group individually: *adv2*, *adv3*, *adv4*, *adv5*. Additionally, because *Graph CSR* has the highest number of page faults, we pair it with the next dense two object groups, *Community Info.* and *Others* as *adv6*, *adv7*, and *adv8*. Lastly, since its impact on page faults is negligible, we create *adv9* to test the object groups except *Hash Tables*.

Table 4.5. Configurations for memory advises applied to object groups.

Name	**preferredLocation CPU**	**accessedBy GPU**
base	-	-
adv1	All object groups	All object groups
adv2	Graph's CSR	Graph's CSR
adv3	Hash Tables	Hash Tables
adv4	Community Info.	Community Info.
adv5	Others	Others
adv6	Graph's CSR, Community Info.	Graph's CSR, Community Info.
adv7	Graph's CSR, Others	Graph's CSR, Others
adv8	Community Info., Others	Community Info., Others
adv9	Graph's CSR, Community Info., Others	Graph's CSR, Community Info., Others

Note: *CPU* means the host's memory and *GPU* means the device's memory.

4.6. Results

We collected the running times of Rundemanen in the PASCAL and AMPERE environments for several datasets with memory advises at 0%, 10%, 30%, 50%, and 70% artificial oversubscriptions explained in Section 3.2.5.. Figures 4.20., 4.21., and 4.22. display the results along with Table 4.6., presenting the average performance gain of each memory advice over the *base* configuration.

The running time of *base* shows an increasing trend as the oversubscription rate increases because, when there is memory oversubscription, we cannot mitigate page thrashing by directly accessing data from the host memory. Additionally, we cannot obtain performance benefits from the pages of the memory addresses exhibiting higher temporal or spatial locality because these pages are also subject to eviction, alongside other pages. *adv3* and *adv4* exhibit similar behavior to *base*, as the contribution of *Hash Tables* is negligible, and the contribution of *Community Info.* is so small to the page faults(see Figure 4.10.) compared to other object groups. Thus, applying memory advice to these two object groups provides almost no benefit over *base*.

Applying memory advises to all object groups as in *adv1* yields consistently average performance across all datasets. The running time remains unaffected by the increasing oversubscription rate as the data reside in the host memory and are accessed from that location, irrespective of the available device memory.

Graph CSR is the main contributor to page faults followed by *Others*. As a result, the memory advises applied to these object groups, *adv2* and *adv7*, enhance the *base*'s performance a lot when there is oversubscription. At 70% oversubscription, *adv7* has the most performance gain, and at 50% oversubscription, either *adv2* or *adv7* shows the most performance gain, with only one exception for rgg_n_2_24_s0 in the PASCAL environment. Furthermore, at 30% oversubscription, *adv2*, if the best is neither *base* nor *adv5*, outperforms the performance of other memory advises.

adv6, *adv8* and *adv9* adversely affect performance at smaller oversubscriptions except for audikw_1, and cage15; at higher oversubscriptions, their performance gain falls somewhere in the middle compared to other memory advises. Although the performance gain of *adv5* is not significant for all datasets, we observe that it shows good performance gain at 10%, 30%, and 50% oversubscriptions for rgg_n_2_24_s0 and cage15.

As a side note, although the application performance is better in the AMPERE environment than in the PASCAL environment, the performance gain is negligible in contrast to the gains obtained when applying memory advises.

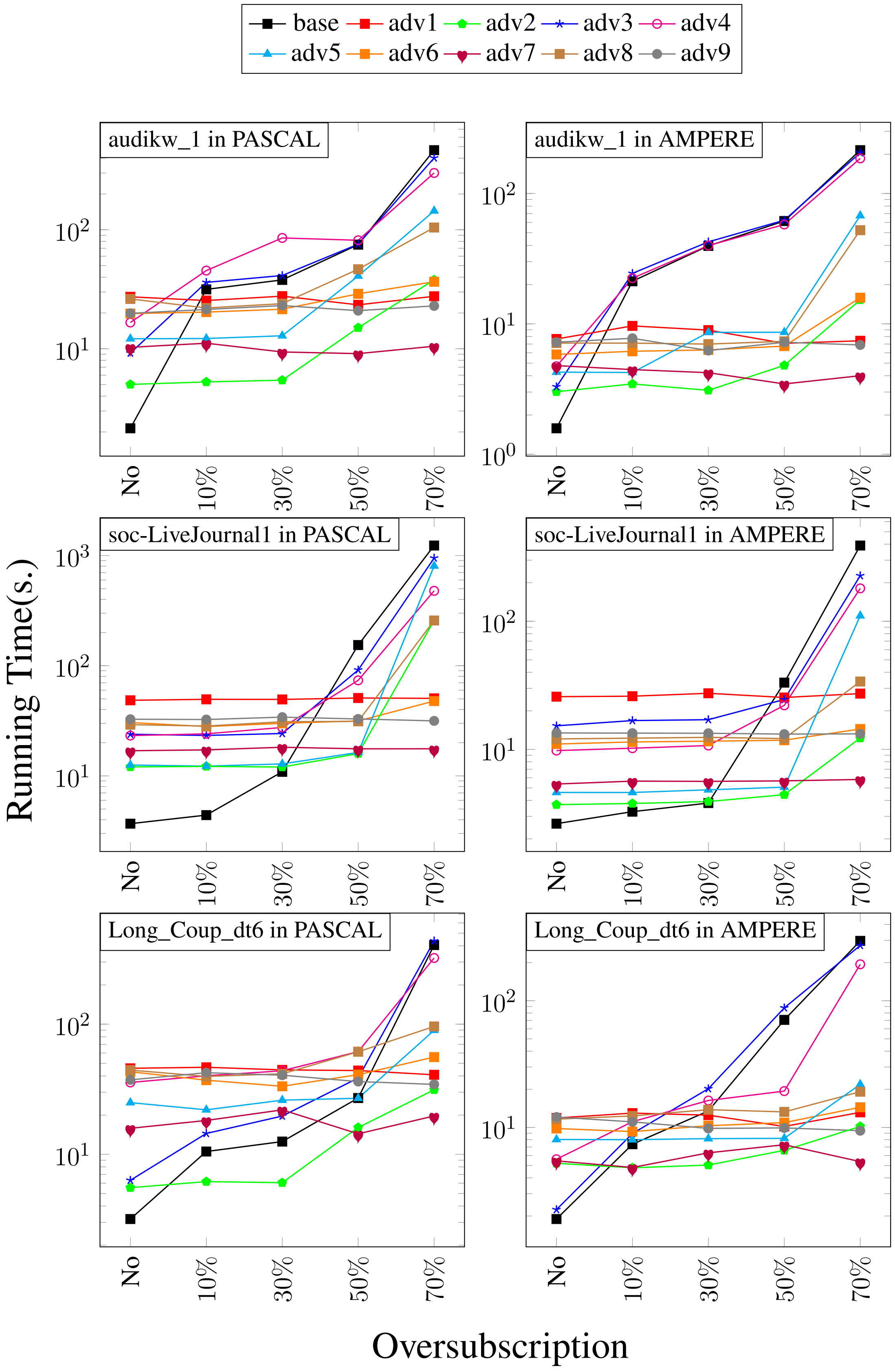


Figure 4.20. The running times of Rundemanen for several datasets with memory advises at different oversubscription rates (I).

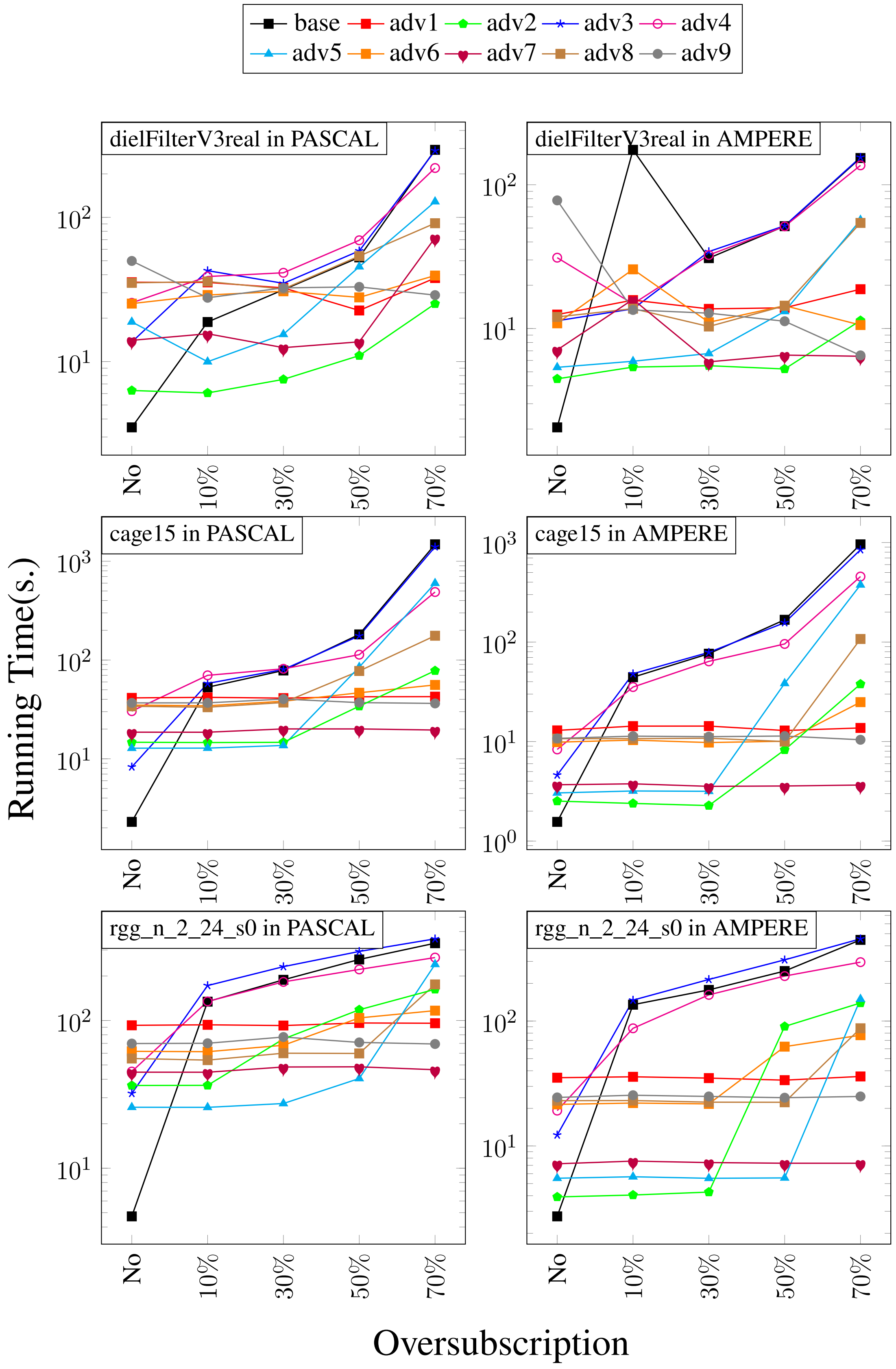


Figure 4.21. The running times of Rundemanen for several datasets with memory advises at different oversubscription rates (II).

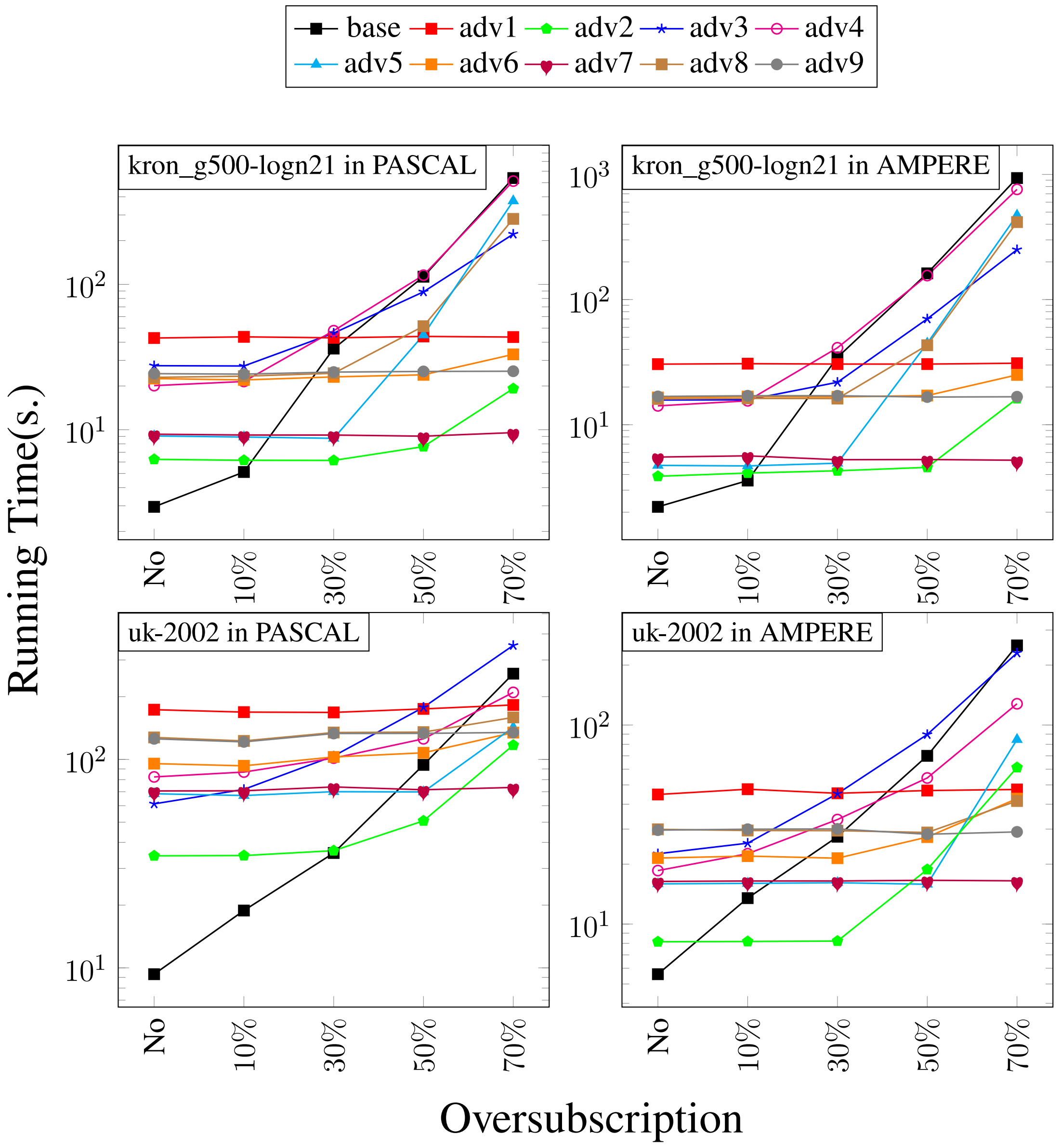


Figure 4.22. The running times of Rundemanen for several datasets with memory advises at different oversubscription rates (III).

When we divide the running time of each advice by the running time of *base* and calculate the average, we obtain Table 4.6.. As the oversubscription rate increases, all advises' average performance gains also increase except *adv3*'s and *adv4*'s at 30% oversubscription.

Table 4.6. The average performance gains of all advises over *base*.

Advise	**Oversubscription**				
	No	**10%**	**30%**	**50%**	**70%**
adv1	0.098	1.642	1.742	4.194	18.934
adv2	0.470	**7.201**	**8.583**	9.274	17.004
adv3	0.278	1.351	0.783	1.069	1.333
adv4	0.171	1.323	0.754	1.319	1.629
adv5	0.312	5.687	6.266	6.854	3.260
adv6	0.149	1.833	2.518	4.790	16.512
adv7	0.284	3.778	5.702	**13.020**	**62.516**
adv8	0.131	2.129	2.387	4.478	5.429
adv9	0.117	2.047	2.290	5.272	27.527

4.6.1. Results of Non-fitting Datasets

The experiments whose results are shown above are run with artificial oversubscription scenarios. These scenarios might not completely represent the reality. To ensure that the results are reliable, we also test some datasets that already do not fit into the GPU's memory in the AMPERE environment. Table 4.7. shows these datasets and how much percentage of them are oversubscribed.

Table 4.7. The Oversubscription rates of datasets collected in the AMPERE environment.

Dataset	**Peak Memory Requirement**	**Oversub. Ratio**
uk-2005	13.7GB	37%
kmer_V1r	14.7GB	41%
kron_24_24	15.9GB	45%
mawi_20151202033	20.9GB	58%
sk-2005	34.9GB	75%

According to the chart in Figure 4.23., as in artificial oversubscription scenarios above, *adv2* and *adv7* have the shortest running times. In contrast to the results of the artificial oversubscription scenarios above, *adv2*'s performance gain overscores *adv7* for all datasets regardless of the oversubscription percentage.

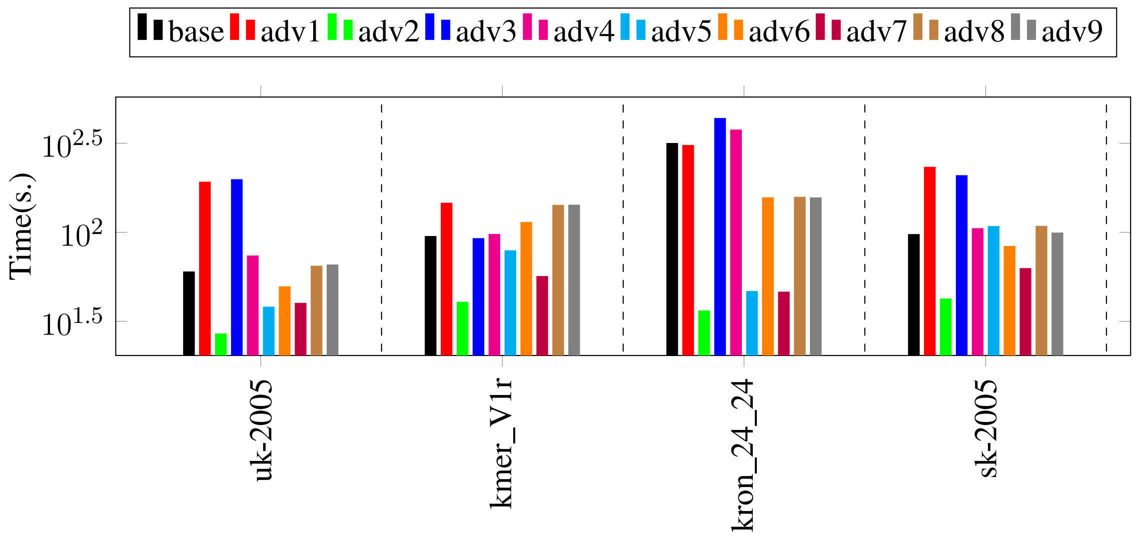


Figure 4.23. The running times of the application for non-fitting datasets collected in the AMPERE environment.

4.7. Discussion

Our study's findings indicate a notable performance gain when we make the object group with the highest number of page faults to be accessed directly from the main memory. By applying memory advises to the graph's data structures ()*Graph's CSR*), we achieved performance improvements of 7x, 8x, 9x, and a substantial 17x at 10%, 30%, 50%, and 70% artificial oversubscriptions, respectively, compared to the naive version. Extending memory advises to include temporary structures (*Others*) resulted in a significant 62x performance gain at a 70% artificial oversubscription. These outcomes align with our expectations, supporting the efficacy of our proposed memory optimization strategies.

There are some limitations to this work. Firstly, the current version applies memory advises in a fixed way for all datasets, indicating a lack of adaptability. Additionally, the absence of a clear explanation for the outcomes linked to specific memory suggests a need for deeper investigation into the underlying mechanisms at play. Another noteworthy limitation is the manual effort in generating the charts, presenting a practical challenge highlighting the importance of refining data collection methods in future studies. These identified limitations underscore the nuances in the current research and point toward areas for improvement and growth.

In future research, memory advises could be adjusted adaptively according to the data characteristics, which may improve the performance. These memory advises could also be tested for other graph applications to see if similar benefits are brought. Furthermore, a framework can be developed that automatically collects memory accesses and page faults and then generates charts.

CHAPTER 5

CONCLUSION

Running a GPU application with non-fitting datasets is straightforward using *thrust*'s vector, but without fine-grained memory optimization, performance degrades. Unified Memory allows us to modify data access with memory advises, but doing so without understanding the application's memory access characteristics can harm performance.

Following our analysis of Rundemanen, we identified that data structures related to graph representation significantly contribute to page faults, particularly at higher oversubscription rates, negatively impacting performance regardless of the dataset characteristics.

By applying memory advises solely to the graph's representation data structures, we enable the GPU to access related data from host memory directly, reducing memory thrashing. This led to a substantial performance gain over the naive version at higher oversubscription percentages. Additionally, we recognized that the performance benefit of applying memory advises to other data structures is highly dependent on the specific characteristics of the datasets.

Examining performance outcomes, on average, we achieved 7x, 8x, 9x, and 17x performance gains at best over the naive version at 10%, 30%, 50%, and 70% oversubscriptions by applying memory advises to the graph's data structures (*Graph's CSR*). Notably, extending memory advises by applying it to also temporary structures (*Others*) resulted in a remarkable 62x performance gain at 70% oversubscription.

In summary, our exploration of Rundemanen with large datasets underscores the critical role of fine-grained memory optimization. The strategic use of targeted memory advises is crucial in achieving significant performance improvements, particularly in scenarios involving oversubscription.

REFERENCES

Alseqyani, Abdulrahman, and Abdullah Almutairi. 2023. “History and Future Trends of Multicore Computer Architecture.” *International Journal of Computer Graphics and Animation (IJCGA)* (Saudi Arabia) 13 (2).

Bader, David A., Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. 2014. “Benchmarking for Graph Clustering and Partitioning.” In *Encyclopedia of Social Network Analysis and Mining.* https://api.semanticscholar.org/CorpusID:44907708.

Beamer, Scott, Krste Asanovic, and David A. Patterson. 2015. “The GAP Benchmark Suite.” *CoRR* abs/1508.03619. arXiv: 1508.03619. http://arxiv.org/abs/1508.03619.

Benson, Dennis A, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. 2012. “GenBank” [in en]. *Nucleic Acids Res* (England) 41, no. Database issue (November): 36–42.

Bergamaschi, Luca, Massimiliano Ferronato, and Giuseppe Gambolati. 2007. “Novel preconditioners for the iterative solution to FE-discretized coupled consolidation equations.” *Computer Methods in Applied Mechanics and Engineering* 196 (25): 2647–2656. ISSN: 0045-7825. https://doi.org/https://doi.org/10.1016/j.cma.2007.01.013. https://www.sciencedirect.com/science/article/pii/S0045782507000266.

Bergamaschi, Luca, Massimiliano Ferronato, and Giuseppe Gambolati. 2008. “Mixed Constraint Preconditioners for the iterative solution of FE coupled consolidation equations.” *Journal of Computational Physics* 227 (23): 9885–9897. ISSN: 0021-9991. https://doi.org/https://doi.org/10.1016/j.jcp.2008.08.002. https://www.sciencedirect.com/science/article/pii/S002199910800421X.

Blondel, Vincent D, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. “Fast unfolding of communities in large networks.” *Journal of Statistical Mechanics: Theory and Experiment* 2008, no. 10 (October): P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008.

Boldi, Paolo, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. “UbiCrawler: a scalable fully distributed Web crawler.” *Software: Practice and Experience* 34 (8): 711–726. https://doi.org/https://doi.org/10.1002/spe.587. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.587. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.587.

Boldi, Paolo, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. “Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks.” In *Proceedings of the 20th international conference on World Wide Web,* edited by Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, 587–596. ACM Press.

Boldi, Paolo, and Sebastiano Vigna. 2004. "The WebGraph Framework I: Compression Techniques." In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004),* 595–601. Manhattan, USA: ACM Press.

Burtscher, Martin, Rupesh Nasre, and Keshav Pingali. 2012. "A quantitative study of irregular programs on GPUs." In *2012 IEEE International Symposium on Workload Characterization (IISWC),* 141–151. https://doi.org/10.1109/IISWC.2012.6402918.

Davis, Timothy A., and Yifan Hu. 2011. "The University of Florida Sparse Matrix Collection." *ACM Trans. Math. Softw.* (New York, NY, USA) 38, no. 1 (December). ISSN: 0098-3500. https://doi.org/10.1145/2049662.2049663. https://doi.org/10.1145/2049662.2049663.

Dziekonski, Adam, Adam Lamecki, and Michal Mrozowski. 2011. "Tuning a Hybrid GPU-CPU V-Cycle Multilevel Preconditioner for Solving Large Real and Complex Systems of FEM Equations." *IEEE Antennas and Wireless Propagation Letters* 10:619–622. https://doi.org/10.1109/LAWP.2011.2159769.

Ferronato, Massimiliano, Luca Bergamaschi, and Giuseppe Gambolati. 2010. "Performance and robustness of block constraint preconditioners in finite element coupled consolidation problems." *International Journal for Numerical Methods in Engineering* 81 (3): 381–402. https://doi.org/https://doi.org/10.1002/nme.2702. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2702. https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2702.

Gawande, Nitin, Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2022. "Towards scaling community detection on distributed-memory heterogeneous systems." *Parallel Computing* 111:102898. ISSN: 0167-8191. https://doi.org/https://doi.org/10.1016/j.parco.2022.102898. https://www.sciencedirect.com/science/article/pii/S0167819122000060.

Han, Tianyi David, and Tarek S. Abdelrahman. 2011. "Reducing Branch Divergence in GPU Programs." In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units.* GPGPU-4. Newport Beach, California, USA: Association for Computing Machinery. ISBN: 9781450305693. https://doi.org/10.1145/1964179.1964184. https://doi.org/10.1145/1964179.1964184.

Heukelum, A. van, G.T. Barkema, and R.H. Bisseling. 2002. "DNA Electrophoresis Studied with the Cage Model." *Journal of Computational Physics* 180, no. 1 (July): 313–326. https://doi.org/10.1006/jcph.2002.7095. https://doi.org/10.1006%2Fjcph.2002.7095.

Holtgrewe, Manuel, Peter Sanders, and Christian Schulz. 2010. "Engineering a scalable high quality graph partitioner." In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS),* 1–12. https://doi.org/10.1109/IPDPS.2010.5470485.

Hong, Sungpack, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. “Accelerating CUDA Graph Algorithms at Maximum Warp.” In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming,* 267–276. PPoPP ’11. San Antonio, TX, USA: Association for Computing Machinery. ISBN: 9781450301190. https://doi.org/10.1145/1941553.1941590. https://doi.org/10.1145/1941553.1941590.

Janna, Carlo, Massimiliano Ferronato, and Giuseppe Gambolati. 2012. “Parallel inexact constraint preconditioning for ill-conditioned consolidation problems.” *Computational Geosciences* 16, no. 3 (June): 661–675.

John Cheng, Ty McKercher, Max Grossman. 2014. *Professional CUDA C Programming.* 1st. GBR: Wrox Press Ltd. ISBN: 1118739329.

Jurij Leskovec, Jon Kleinberg, Deepayan Chakrabarti, and Christos Faloutsos. 2005. “Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication.” In *Knowledge Discovery in Databases: PKDD 2005,* 133–145. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-31665-7.

Kuroda, T. 2001. “CMOS design challenges to power wall.” In *Digest of Papers. Microprocesses and Nanotechnology 2001. 2001 International Microprocesses and Nanotechnology Conference (IEEE Cat. No.01EX468),* 6–7. https://doi.org/10.1109/IMNC.2001.984030.

Leskovec, Jure, and Andrej Krevl. 2014. *SNAP Datasets: Stanford Large Network Dataset Collection.* http://snap.stanford.edu/data. (Accessed: November 19, 2023).

Liu, Han, Fan Yang, and Ding Liu. 2016. “Genetic algorithm optimizing modularity for community detection in complex networks.” In *2016 35th Chinese Control Conference (CCC),* 1252–1256. https://doi.org/10.1109/ChiCC.2016.7553259.

Lü, Zhipeng, and Wenqi Huang. 2009. “Iterated tabu search for identifying community structure in complex networks.” *Physical review. E, Statistical, nonlinear, and soft matter physics* 80 (August): 026130. https://doi.org/10.1103/PhysRevE.80.026130.

Mahantesh Halappanavar, Ananth Kalyanaraman, Howard (Hao) Lu, and Sayan Ghosh. 2020. *grappolo.* https://github.com/ECP-ExaGraph/grappolo. (Accessed: Novemner 26, 2022).

Mayer, S. 2004. *UF Sparse Matrix Collection, Tim Davis.* http://www.cise.ufl.edu/research/sparse/matrices/GHS_psdef/audikw_1. (Accessed: June 22, 2023).

Moore, Gordon E. 2006. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” *IEEE Solid-State Circuits Society Newsletter* 11 (3): 33–35. https://doi.org/10.1109/N-SSC.2006.4785860.

Naim, Md., Fredrik Manne, Mahantesh Halappanavar, and Antonino Tumeo. 2017. “Community Detection on the GPU.” In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS),* 625–634. https://doi.org/10.1109/IPDPS.2017.16.

Newman, M. E. J. 2004. “Fast algorithm for detecting community structure in networks.” *Physical Review E* 69, no. 6 (June). https://doi.org/10.1103/physreve.69.066133.

Newman, M. E. J., and M. Girvan. 2004. “Finding and evaluating community structure in networks.” *Physical Review E* 69, no. 2 (February). https://doi.org/10.1103/physreve.69.026113.

NVIDIA. 2023a. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (Accessed: February 17, 2023).

NVIDIA. 2023b. *thrust.* https://github.com/NVIDIA/thrust. (Accessed: June 13, 2023).

Owens, John D., David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. “A Survey of General-Purpose Computation on Graphics Hardware.” *Computer Graphics Forum* 26 (1): 80–113. https://doi.org/https://doi.org/10.1111/j.1467-8659.2007.01012.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x. https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01012.x.

Peleg, A., and U. Weiser. 1996. “MMX technology extension to the Intel architecture.” *IEEE Micro* 16 (4): 42–50. https://doi.org/10.1109/40.526924.

RAPIDS.ai. 2022. *cuGraph - RAPIDS Graph Analytics Library.* https://github.com/rapidsai/cugraph. (Accessed: April 3, 2023).

Shao, Chuanming, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. “Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions.” In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering,* 67–75. ICPE ’22. Beijing, China: Association for Computing Machinery. ISBN: 9781450391436. https://doi.org/10.1145/3489525.3511691. https://doi.org/10.1145/3489525.3511691.

Shi, Xuanhua, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. “Graph Processing on GPUs: A Survey.” *ACM Comput. Surv.* (New York, NY, USA) 50, no. 6 (January). ISSN: 0360-0300. https://doi.org/10.1145/3128571. https://doi.org/10.1145/3128571.

Su, Xing, Shan Xue, Fanzhen Liu, Jia Wu, Jian Yang, Chuan Zhou, Wenbin Hu, et al. 2022. “A Comprehensive Survey on Community Detection With Deep Learning.” *IEEE Transactions on Neural Networks and Learning Systems,* 1–21. https://doi.org/10.1109/tnnls.2021.3137396.

Tasgin, Mursel, Amac Herdagdelen, and H. Bingol. 2006. “Community Detection in Complex Networks Using Genetic Algorithms.” *arXiv: Physics and Society,* https://api.semanticscholar.org/CorpusID:265748539.

Venu, Balaji. 2011. “Multi-core processors - An overview.” *ArXiv* abs/1110.3535. https://api.semanticscholar.org/CorpusID:16243328.

Wikipedia. 2023. *CUDA.* http://en.wikipedia.org/w/index.php?title=CUDA. (Accessed: November 4, 2023).