Dynamic Buffer Management in Massively Parallel Systems: A Case on GPUs

Minh Pham*, Hao Li*, Yongke Yuan†, Chengcheng Mou*,
Kandethody Ramachandran*, Zichen Xu‡ and Yicheng Tu*
*Department of Computer Science and Engineering, University of South Florida
4202 E Fowler Ave., ENB 118, Tampa, FL, U.S.A.
{minhpham, haoli1, chengcheng, ram, tu}@mail.usf.edu
†School of Economics and Management, Beijing University of Technology
100 Pingleyuan, Chaoyang District, Beijing 100124, China
yyk@bjut.edu.cn
‡Jiaxing Yunbao Scientific Co. Ltd.
3339 Linggongtang Rd, Suite 379, Nanhu District, Jiaxing, Jiangsu, China
zichenxu@outlook.com

***Abstract*—Massively parallel systems, such as Graphics Processing Units (GPUs), are becoming increasingly important in today's data-intensive computing environments. Due to the high level of parallelism, there are unique challenges in developing system software on massively parallel hardware to efficiently support a large number of parallel threads. One such challenge is designing a dynamic memory allocator whose task is to allocate memory chunks to requesting threads at runtime. A traditional design of a memory allocator involves maintaining a global data structure, such as a list of free pages. However, the centralized data structure can easily become a bottleneck in a massively parallel system. The bottleneck still exists when multiple queues are maintained, as done in state-of-the-art GPU memory allocation solutions. In this paper, we present a novel approach for designing dynamic memory allocation without a centralized data structure. At runtime, the threads follow a random search procedure to locate free pages. We develop mathematical models to demonstrate that our methods achieve asymptotically lower latency than the traditional queue-based design. Extensive experiments show consistency to our mathematical models and demonstrate that our solutions can achieve up to two orders of magnitude improvement in latency over the best-known existing solutions.**

I. Introduction

Recent development in the semiconductor industry features an increasing number of processing cores on a chip, resulting in massively parallel computing capability. For example, the latest CPU products encapsulate up to 64 cores in one chip (e.g., AMD Ryzen 3990X) [1]. The co-processor world goes to an extreme on that by integrating thousands of thin cores into one processor, with a salient example being modern Graphics Processing Units (GPUs).

GPUs have become an indispensable component in today's high-performance computing (HPC) systems and have shown great value in many compute-intensive applications. In addition, there is also a strong movement of developing system software on GPUs, such as database management systems. [43], [54], [60], [62], [63] Dynamic memory allocation on GPUs was first introduced about ten years ago by NVIDIA and many other solutions have been proposed since then [59]. Many GPU-based applications benefit from dynamic memory allocation such as graph analytics [11], [58], data analytics [6], [51], and databases [5], [26].

There are unique challenges in developing system software on massively parallel hardware, mostly imposed by the need to support a large number of parallel threads efficiently and the architectural complexity of the GPU hardware. Dynamic memory allocators in particular face challenges such as thread contention and synchronization overhead, and multiple studies [59] have proposed solutions to address these challenges. Similar to traditional memory allocators, such solutions utilize a shared data structure to keep track of available memory units [59]. For example, the current state-of-the-art solution, Ouroboros [57] uses a combination of linked-lists, arrays, and queues to reduce thread contention and memory fragmentation and was shown to outperform previous solutions in a recent comparative study [59]. Nevertheless, we show in section II that thread contention, synchronization overhead, and memory overhead are still problematic with Ouroboros in many use cases. In this paper, we argue that dynamic memory allocators demand a complete rethinking in their designs such that: (1) global states are avoided and (2) the search for free memory units is done through statistical processes.

GPUs are designed to be high-throughput systems – its performance depends on running a large number of parallel threads. A modern CPU could run tens of threads simultaneously while it is common to see tens of thousands of active threads in a GPU. This demands us to take a second look at the classic design of a system's memory manager. Specifically, traditional memory managers maintain a global state (e.g., head and tail of a queue) to keep track of available memory units. The MALLOC and FREE operations have to access such states in a protected manner. The protection can be done via a software *lock* (e.g., mutex), with a latency at the hundred-millisecond level on CPU-based systems. [18], [35] Hardware-supported mechanisms called *atomic* operations are widely

used to relieve such a bottleneck. However, while used in GPUs, this strategy still carries excessively high overhead. Although fast, atomic operations have to be executed sequentially in case of conflicts – the large number of concurrent threads in GPUs leads to a long waiting queue in atomic access to global states.

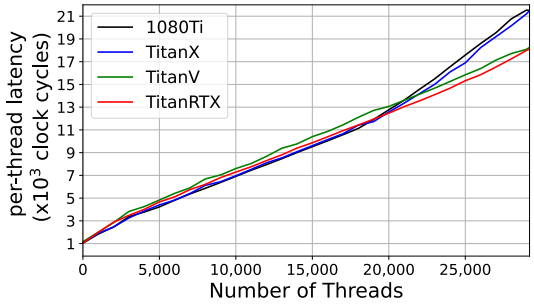

Fig. 1. Average latency in accessing a global variable via atomic operations in different NVidia GPUs.

Figure 1 reports the average latency of performing an atomic operation against one global 32-bit integer under varying number of concurrent (active) threads. Clearly, the average latency per thread grows linearly with the number of concurrent threads. Furthermore, the latency is much higher than that of CPUs. For example, we run the same code on an AMD EPYC 7662 CPU, the latency is only 23 clock cycles for one thread and 1,262 clock cycles for 128 threads.

In this paper, we present a high-performance memory management framework for massively parallel hardware such as GPUs. Unlike traditional wisdom that involves global states, this is a fundamentally new solution that carries very little overhead in allocating memory and is almost free for releasing memory. Instead of keeping any global states explicitly, we let the threads statistically infer the locations of available memory units via a random algorithm. We develop analytical models to demonstrate that our method achieves asymptotically shorter latency than the state-of-the-art GPU memory allocators.

We also report a number of techniques to further improve the performance of the random method. Specifically, we present the use of bitmap to reduce the number of expected steps needed to find a free page by a factor of 32 or 64; a page sharing mechanism among neighboring threads that essentially minimizes resource waste due to code divergence. Based on those two techniques, we also develop an algorithm for serving requests of multiple pages. The performance advantage of our solutions are fully supported by extensive experiments. In particular, in a unit-test environment, our solution was found to deliver a speedup of up to two orders of magnitude over the best existing solutions. To the best of our knowledge, our development is the first GPU solution that achieves 10-microsecond level latency in memory allocation.

Paper Organization: The remainder of this paper is organized as follows: Section II sketches the background that includes the technical foundation, the current state-of-the-art, and its drawbacks; Section III introduces our memory management framework, key algorithms, and the mathematical reasoning for our framework; Section IV presents advanced techniques with improved performance; Section V shows results of experimental evaluation (unit-tests) in comparison with the state-of-the-art; Section VII surveys more relevant literature; and Section VIII concludes this paper.

II. Background

A. GPU Architecture

Although our work can be applied to other parallel hardware, we focus on NVidia GPUs and the associated CUDA programming language. The architecture of a typical NVidia GPU is shown in Figure 3. A GPU card has multiple Streaming Multiprocessors (SM) that each consists of tens of processing cores. Each SM also contains a register pool (e.g., 256 KB) and *shared memory* similar to L1 cache up to 96KB, which are both strictly accessible only by threads running on that SM. Starting at Volta architecture [39], L1 cache and shared memory are merged together to provide a larger cache for an SM. On board the GPU there are a certain amount (e.g., 24GB for Titan RTX) of *global memory*, which can be accessed in parallel by cores in different SMs. The bandwidth of global memory can be as high as 1555 GB/s [52]. Global memory communicates with main computer (host) memory through PCIe with a bandwidth up to 32 GB/s [36].

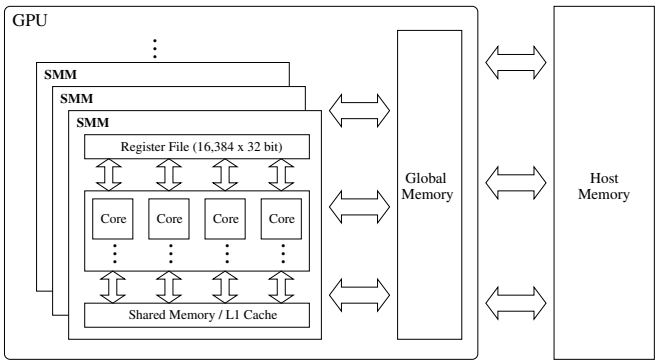

Fig. 2. Architecture of a modern NVidia GPU

In the CUDA programming model, a function executed in parallel on the GPU is called a CUDA *kernel*. A kernel can be launched with a large number (e.g., millions) of concurrent *threads*. We call all the threads in the kernel a *grid*, and the grid is further divided into multiple *blocks*. Each block contains the same number of threads and a block will be bound to a single SM for execution. Within a block, CUDA always schedules 32 threads in a group called *warp* together for SIMD-style execution. Thus, a warp : (1) is the basic execution unit in CUDA; and (2) code divergence affects performance due to the SIMD execution model. Although we can launch threads in the millions, there are limited resources on board therefore the thread blocks will have to take turns to be executed. In other words, the maximum parallelism is achieved once we launch the kernel with thread counts beyond tens of thousands [34].

B. Dynamic Memory Allocation in CPU-based Systems

Memory allocators on CPUs have been well studied since the 1960s [56]. Some of the most popular mechanisms include

Sequential Fits (a single linked-list of all free pages), Segregated Free Lists, Buddy Systems (multiple memory pools of power-of-two in size), Indexed Fits (page information indexed in arrays), and Bitmapped Fits (page information indexed in bitmaps). Popular implementations are the GNU malloc [21] and the Hoard malloc [8], both of which use multiple arenas for concurrent processing.

C. Dynamic Memory Allocation in GPU-based Systems

In CUDA, we often pre-allocate a certain amount of global memory (via CUDAMALLOC function) to serve all runtime memory needs of a GPU kernel. However, memory consumption is unknown beforehand in many applications. This renders either over-allocation or terminating the kernel due to lack of memory. The typical approach [27] to deal with this problem is to run the task twice: the first run is only for calculating the output size, then the output memory can be precisely allocated, and the second run will finish the task. This obviously carries unnecessary overhead. Thus, a major challenge on GPU systems is to dynamically allocate device memory for output results without interrupting kernel execution. In 2009, NVidia released a dynamic memory allocator for CUDA [40]. That started a series of efforts on this topic, including XMalloc (2010) [28], ScatterAlloc (2012) [48], FDGMalloc (2013) [55], HAlloc (2014) [3], Reg-Eff (2015) [53], DynaSOAr (2019) [47], and Ouroboros (2020) [57]. A recent comparative study [59] showed that Ouroboros outperformed all aforementioned methods in both allocation performance and space efficiency and thus can be considered the state-of-the-art.

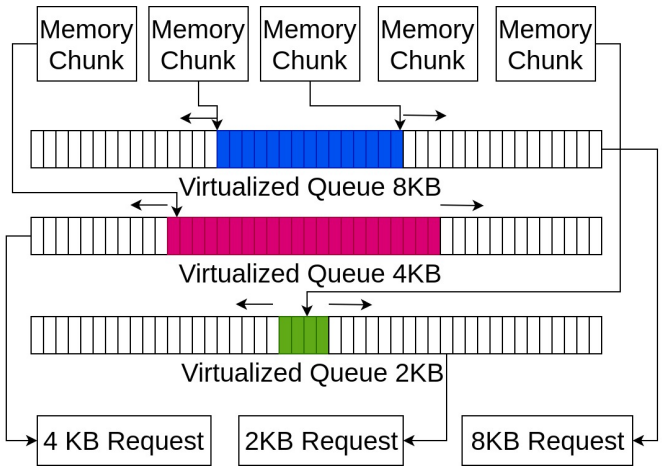

Fig. 3. Ouroboros Design: memory chunks are used to extend virtualized queues upon allocation requests. Multiple queues are maintained, each serving requests of different sizes

Similar to the Buddy Systems mechanism on CPU implementations, Ouroboros divides the managed memory region into multiple queues, each serving a page size twice as large as that of a previous queue. Instead of pre-allocating memory for the queues, the concept of *Virtualized Queues* was introduced. The main idea is that queues are dynamically stored on pages in the pool and a new large page (chunk) is only allocated to a queue when it actually needs more space. In this design, the authors avoided pre-allocating memory for all the queues.

As a queue-based design, Ouroboros suffers from the long latency in accessing queue states concurrently by using the bulk semaphore described in [20]. Furthermore, Ouroboros' design creates significant memory overhead when request sizes are not close to and lower than the pre-determined sizes and significant latency when there are many requests of similar sizes. Figure 4 illustrate these issues. In this experiment, we ran 5 programs with Ouroboros: (1) each thread allocates then frees 4096B, (2) each thread allocates 4096B, (3) each thread allocates 4100B, (4) each thread allocates 8192B, and (5) each thread allocates varying sizes. We ran each program 100 times on varying number of threads and calculate the average kernel time.

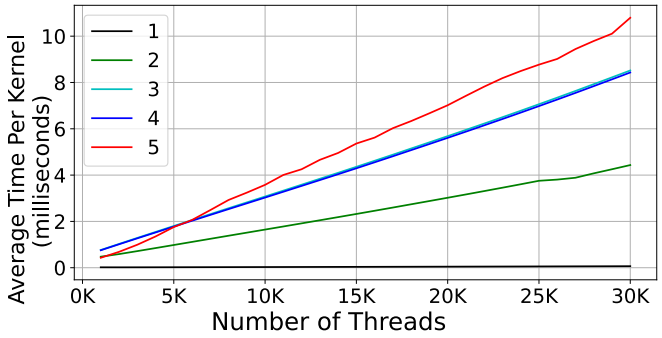

Fig. 4. Ouroboros' Performance under the five scenarios

Scenario (1) has the same setup as in Figure 9 in [59] and our result is consistent with those presented. In this setup, the first iteration takes a significantly long time to allocate new pages for the queues, but after the free operation, the freed pages are put into the populated queue and the second iteration takes no time to get the free pages from the queue. This situation is repeated for the remaining 98 iterations and makes the average kernel time artificially low. In Scenario (2), we only performed allocation without freeing to force Ouroboros to keep extending its virtualized queues. As a result, average kernel time increased by up to 67 times over Scenario (1). Furthermore, when profiling the program with NVIDIA's Visual Profiler, we found that allocating new pages for the queues makes the kernel achieve only 8% warp-efficiency, which is clearly unacceptable. Scenario (3) increases the request size by only 4B and sees the average kernel time double. Scenario (4) has the request size of 8192B but virtually the same average kernel time as in Scenario (3). This proves that the 4100B request size is put in the 8192B queue and wasted 50% of the allocated space. The varying request size in Scenario (5) seems to amplify these problems.

III. PARALLEL MEMORY MANAGEMENT FRAMEWORK

In this section, we present an efficient memory management framework for GPUs. This section is divided into three subsections.

A. Core Idea: Search for free pages by a random process

First, we divide the main memory on GPU into pages of equal size. In the previous studies on CPUs and GPUs, mutex

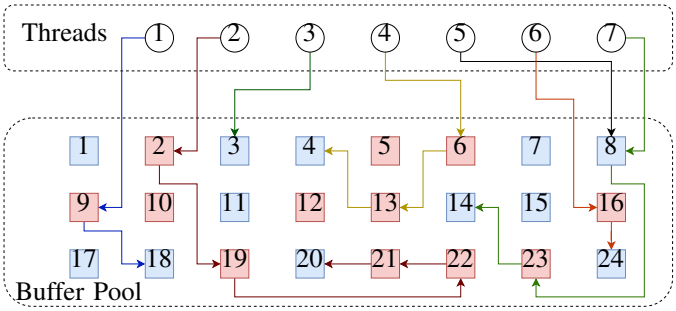

Fig. 5. Demonstration of the RW-based page request algorithm. The path visited by the same thread is colored the same. Blue pages are free, red pages are occupied.

Algorithm 1: GETPAGE based on Random Walk

output: the ID of a free page

```
1: while True do
2:     p ← random integer within [0, T)
3:     if pages[p].used is false then
4:         try to set pages[p].used to true
5:         if Above is a success then
6:             return p
7:         end if
8:     end if
9: end while
```

locks and queues are utilized to avoid Write After Write. A small number of queues and mutex locks can become a bottleneck in a GPU system when tens of thousands of threads make a request at the same time. We need a better design to fully release the parallel computing power of GPUs. To that end, we propose a Random Walk (RW) algorithm that does not depend on any global state to manage page allocation and recycling. Instead of using a few mutex locks or queues on global memory for the entire system, each page will have its own mutex lock, i.e., once the *used* flag in a page is set, it is considered not available. In requesting a page, each thread will generate a random page ID. If the corresponding page is free, the thread will get the page. Otherwise, the thread will generate a new page IDs till it finds one free page. The main idea is: we let all threads act independently and therefore there is no need to wait in a queue for accessing a shared state. This releases the parallel computing power of the GPU to the greatest extent. Figure 5 shows an illustrative example of how seven parallel threads get their free pages. Here the blue squares represent free pages, and red ones represent occupied pages. Each thread essentially repetitively generates random page IDs till it finds a free page.

Detailed implementation of the RW-based algorithm can be found in Algorithm 1. Note that in this paper, all pseudo-code is presented from the perspective of a single thread, reflecting the single-program-multiple-data (SPMD) programming model for modern GPUs.

Even with the drawback of queue-based ideas, the advantage of RW is still counter-intuitive: the traditional queue-based methods allow for $O(1)$ time in finding a free page while it could take many steps for RW. However, although the number of steps to get a free page can be big for some threads (e.g., 5 steps for thread 2), the average number of steps is highly controllable under most scenarios. Our analysis in Section III-B will clearly show this.

Although there are no global variables, acquiring a free page still needs an atomic operation (line 4 of Algorithm 1) because two threads could try to grab the same free page at the same time. For example, in Figure 5, threads 5 and 7 both request page 8 in their first step. However, the atomic operation will only grant access to one thread (e.g., thread 5) and the other threads (e.g., thread 7) will continue its random walk. The above is the only scenario in which two threads can conflict in accessing protected data. Our analysis shows that this scenario will happen with very low probability (Section III-B).

Another great advantage of our method is: the FREEPAGE operation is almost free. Specifically, we only need to clear the *used* bit of the corresponding page, without using any atomic operations.

B. Performance Analysis

Now let us mathematically show how the Random Walk (RW) design is better than the traditional LL-based methods.

1) Metrics for Performance Analysis: In general, latency (of individual threads and all the threads) is an appropriate metrics for evaluating the performance of memory management mechanisms. However, the running time of a CUDA program is affected by many factors, as show in our previous work [34]. Instead, we propose the following two metrics:

1. **Per-Thread Average Steps (TAS)**: the average number of steps taken to find a free page for a thread. In Algorithm 1, this is essentially the average number of iterations executed for the **while** loop;
2. **Per-Warp Average Steps (WAS):** the average of the maximum number of steps taken among all 32 threads within a warp.

Both metrics are directly correlated to latency. While WAS has a stronger correlation with latency than TAS, we achieve more rigorous analysis of the latter. In CUDA, the basic unit of execution is a warp – a group of 32 threads that are scheduled and executed simultaneously by a streaming multiprocessor. The entire warp will hold the computing resources until all threads in it exited. In other words, the latency of a warp is the maximum latency among all 32 threads in the warp. Without detailed knowledge of the CUDA runtime engine, it is non-trivial to develop an accurate model of the running time from the steps of each thread takes. Our previous work [34] shows that, by measuring the average steps a thread takes, we can say the running time is roughly a linear function of per-thread latency. We verified the effectiveness of the two metrics via a large number of experimental runs (details skipped here due to page limit). The results show that the correlation coefficient between TAS and total running time is 0.9046, and that for WAS is 0.962. Following the techniques described in [64], we found that the confidence interval for the difference between the two correlation coefficients is (0.0387, 0.1047). This shows that WAS is a better indicator of total running time.

In the remainder of this paper, we use the following notations in our mathematical analysis :

- For any warp, let $X_i (0 \leq i \leq 31)$ be the random variable representing the number of steps taken until finding a free page. TAS is the expected value of X_i, denoted as $E(X_i)$;
- Let $Y = max(X_i)$ be the max number of steps taken by a thread to find a free page within a warp. WAS is then the expected value of Y, denoted as $E(Y_i)$;
- T is the total number of buffer pages;
- A is the number of available buffer pages;
- N is the total number of concurrent threads.

2) Performance of Queue-Based Solutions: To get started, we can first show performance of GETPAGE and FREEPAGE in a queue-based solution. While one thread takes one step to modify the protected data structure, other threads have to stall for one step. Since N threads request to modify the protected data structure at the same time, the processing queue would have the length of N. We assume that a random thread would have a random position on the processing queue. Then, X_i is uniformly distributed on $[1, N]$. Therefore, TAS is:

$$E(X_i) = \frac{N+1}{2} \tag{1}$$

In order to find $E(Y)$, we first find the cumulative distribution function of Y:

$$\begin{aligned} P(Y \leq y) &= P(X_0 \leq y, X_1 \leq y, ..., X_{31} \leq y) \\ &= P(X_0 \leq y) \cdot P(X_1 \leq y) \cdot ... \cdot P(X_{31} \leq y) \\ &= (\frac{y}{N})^{32} \end{aligned}$$

Then the expectation of Y is:

$$\begin{aligned} E(Y) &= \sum_{k=1}^{N} kP(Y = k) \\ &= \sum_{k=1}^{N} k(P(Y_j > k-1) - P(Y_j > k)) \\ &= \sum_{k=1}^{N} kP(Y_j > k-1) - \sum_{k=1}^{N} kP(Y_j > k) \\ &= \sum_{k=1}^{N} (k-1)P(Y_j > k-1) + \sum_{k=1}^{N} P(Y_j > k-1) - \\ &\quad \sum_{k=1}^{N} kP(Y_j > k) \\ &= \sum_{k=0}^{N} kP(Y_j > k) + \sum_{k=1}^{N} P(Y_j > k-1) - \\ &\quad \sum_{k=1}^{N} kP(Y_j > k) \\ &= 0P(Y_j > 0) + \sum_{k=1}^{N} P(Y_j > k-1) \\ &= \sum_{k=1}^{N} P(Y_j > k-1) \\ &= \sum_{k=1}^{N} (1 - P(Y \leq k-1)) \\ &= \sum_{k=0}^{N} (1 - P(Y \leq k)) \\ &= \sum_{k=0}^{N} (1 - (\frac{k}{N})^{32}) = N - \sum_{k=0}^{N} \frac{k^{32}}{N^{32}} \end{aligned}$$

This can be approximated by the Faulhaber's [31] formula as:

$$\begin{aligned} E(Y) &= N - \frac{\frac{N^{33}}{33} + \frac{N^{32}}{2} + \frac{8N^{31}}{3} - \frac{124N^{29}}{3} + ...}{N^{32}} \\ &= N - \frac{N}{33} + \frac{1}{2} + \frac{8}{3N} - \frac{124}{3N^2} + ... \approx \frac{32}{33}N \end{aligned} \tag{2}$$

Both metrics are linear to N. This is consistent with the results shown in Figure 1. Maintaining multiple queues will not wipe out the issue, it is easy to show that both TAS and WAS are still linearly related to N.

3) Analysis of TAS: The process of acquiring a free page by N parallel threads can be viewed as N parallel series of Bernoulli trials. If there is only one thread requesting a page, its Bernoulli trials have a constant probability of success. When there are multiple threads, a thread's Bernoulli trials will have a decreasing probability of success over time.

To simplify the discussion, we treat the N series of Bernoulli trials as if they are performed **sequentially**, i.e., one only starts after another has finished, and still achieve the same results as the parallel process. This treatment is safe

because of two reasons. First, two parallel threads can totally be performed sequentially if they do not cross path. Second, if two threads cross path, the outcome should still be the same as in the sequential case. For example, in Figure 5, thread 5 and thread 7 cross path at page 8, and the outcome is the same as if thread 7 starts executing after thread 5. Furthermore, we shall prove that the chance of two threads' visiting a free page at the same time is rare.

Before the first thread executes, there are A free pages out of the total T pages. Therefore, the number of steps that the first thread takes until finding a free page, X_0, follows a geometric distribution with $p = A/T$. Therefore,

$$E(X_0) = 1/p = T/A$$

After the first thread finishes and before the second thread executes, there are only $A-1$ pages free. Therefore, the number of steps taken until finding a free page, X_1, follows a geometric distribution with $p = (A-1)/T$. Therefore,

$$E(X_1) = 1/p = T/(A-1)$$

Generalizing the above, the average number of steps taken across all N threads is

$$E(X_i) = \frac{1}{N}\sum_{j=0}^{N-1}\frac{T}{A-j} = \frac{T}{N}\sum_{j=0}^{N-1}\frac{1}{A-j} = \frac{T}{N}(H_A - H_{A-N})$$

where $H_n = \sum_{k=1}^{n}\frac{1}{k}$ is the harmonic series.

We use the Euler-Mascheroni constant [10] to approximate the harmonic series $H_n \approx \gamma + \ln n$. The expected average number of steps is then approximated by

$$\begin{aligned} E(X_i) &\approx \frac{T}{N}[(\gamma + \ln A) - (\gamma + \ln(A-N))] \\ &= \frac{T}{N} ln\left(\frac{A}{A-N}\right) \end{aligned} \tag{3}$$

Unlike the queue-based solution with latency linear to N (Eqs. (1) and (2)), Eq. (3) tells us that the value grows very little with the increase of N. Specifically, under a wide range of N values, the item $ln(\frac{A}{A-N})$ increases very slowly (in a logarithmic manner), and the increase of $E(X_i)$ will be further offset by the inverse of N. The only situation that could lead to a high number of steps is when $A \approx N$, i.e., when there are barely enough pages available for all the threads.

Eq. (3) has a linear growth with the increase of T, but in practice, a larger T value also leads to an increase in A, which would offset the growth by decreasing the logarithmic term.

The above analysis can be verified in Figure 6(a) where we plot the value of formula (5) under different A and N values with $T = 1M$. We chose five different A values , which correspond to $50\%, 10\%, 1\%, 0.7\%$, and 0.5% of total pages T. Note that the case of 0.5% is an extreme scenario – when $N = 5,000$, there is only one page available for each thread – yet the $E(X_i)$ values we calculated are still much lower than that of the queue-based method.

Independence Among Threads: In the above analysis, we made an implicit assumption that all the threads conduct the random walk independently, i.e., their paths do not cross. However, it is possible that multiple threads probe one free page and try to atomically acquire it at the same time. Now we show that the number of such collisions is extremely small.

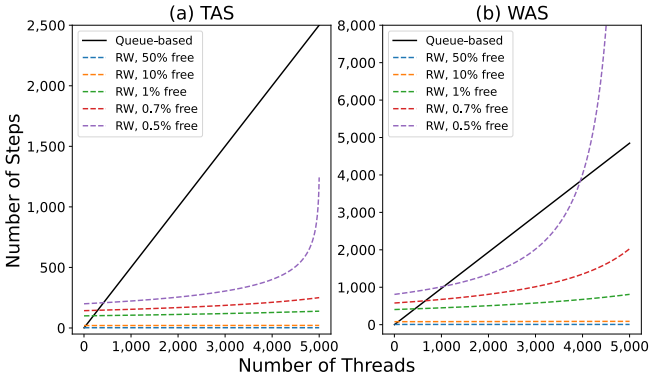

Fig. 6. Change of TAS (a) and WAS (b) values under different N and A values of the RW-based algorithm in comparison to that of Queue-based solution

This situation is analogous to the well-studied *Birthday Paradox* problem: we have T birthdays (pages) and N people (threads). In particular, the expected number of collisions is $\frac{N(N-1)}{2T}$ (see Eq. (2) in [44]). Here by "collision" we refer to two threads' getting the same p value (line 2 of Algorithm 1) at the same time. This number should be small in any reasonable setup: with tens of GBs of global memory and thousand-level parallelism in a modern GPU, we can safely assume $N << T$. For example, with 1 million total pages and 5,000 concurrent threads, the expected number of collisions is only 12.5. Furthermore, performance penalty due to atomic operations exists only when the p-th page is free (i.e., line 4 of Algorithm 1). Therefore, the expected number of collisions is further bounded by

$$\frac{N(N-1)}{2T} \times \frac{A}{T}$$

4) Analysis of WAS: Deriving a closed-form for $E(Y)$ is difficult, but we can find an upper bound of $E(Y)$ as follows. We observe that during the process of N threads' each getting a page, the probability of finding a free page at any moment in the process is at least $\frac{A-N}{T}$. The reason is that A is in the $[A-N, A]$ range during the process. Therefore, $E(X_i)$ is upper bounded by $E(X_i')$ where X_i' follows a Geometric distribution with probability $p = \frac{A-N}{T}$. With that, the cumulative distribution function of X_i' is:

$$P(X_i' \le x) = 1 - (1-p)^x = 1 - \left(\frac{T-A+N}{T}\right)^x$$

Since $E(X_i)$ is upper bounded by $E(X_i')$, $E(Y)$ is also upper bounded by $E(Y')$ where $Y' = max(X_i')$. The cumulative distribution function of Y' is:

$$\begin{aligned} P(Y' \le y) &= P(X_0 \le y, X_1 \le y, ..., X_{31} \le y) \\ &= P(X_0 \le y) \cdot P(X_1 \le y) \cdot ... \cdot P(X_{31} \le y) \\ &= \left(1 - \left(\frac{T-A+N}{T}\right)^y\right)^{32} \end{aligned}$$

Similar to the way we derive Eq. (2), the expectation of Y' is:

$$\begin{aligned}
E(Y') &= \sum_{k=1}^{\infty} kP(Y=k) \\
&= \sum_{k=1}^{\infty} k(P(Y'>k-1) - P(Y'>k)) \\
&= \sum_{k=1}^{\infty} kP(Y'_j>k-1) - \sum_{k=1}^{\infty} kP(Y'>k) \\
&= \sum_{k=1}^{\infty} (k-1)P(Y'>k-1) + \sum_{k=1}^{\infty} P(Y'>k-1) - \\
&\qquad \sum_{k=1}^{\infty} kP(Y'>k) \\
&= \sum_{k=0}^{\infty} kP(Y'>k) + \sum_{k=1}^{\infty} P(Y'>k-1) - \\
&\qquad \sum_{k=1}^{\infty} kP(Y'>k) \\
&= 0P(Y'>0) + \sum_{k=1}^{\infty} P(Y'>k-1) \\
&= \sum_{k=1}^{\infty} P(Y'>k-1) \\
&= \sum_{k=1}^{\infty} (1 - P(Y' \le k-1)) \\
&= \sum_{k=0}^{\infty} (1 - P(Y' \le k)) \\
&= \sum_{k=0}^{\infty} \left[1 - \left(1 - \left(\frac{T-A+N}{T}\right)^k\right)^{32}\right]
\end{aligned}$$

Therefore, an upper bound of WAS $E(Y)$ is

$$E(Y) < \sum_{k=0}^{\infty} \left[1 - \left(1 - \left(\frac{T-A+N}{T}\right)^k\right)^{32}\right] \quad (4)$$

In Figure 6(b), we plot the calculated values of the RHS of Eq. (4) with $T = 1M$. Obviously, this bound is larger than $E(X_i)$ (Figure 6(a)) under the same parameters. Same as $E(X_i)$, the bound of $E(Y)$ is still significantly smaller than the LL-based latency, even under small A/T values such as 0.7%. For the extreme case of $A/T = 0.5\%$, we start to see the bound climb higher than the $E(X_i)$ value of the queue-based method. This can be viewed as a drawback of the RW method, and we will address that in Section IV-B.

IV. Advanced Techniques

The basic RW algorithm can be extended in several directions. First, the memory allocation performance of RW deteriorates when the the percentage of free pages is small. This is caused by the large TAS values under a small A/T ratio, and worsened by the gap between TAS and WAS as a result of code divergence. In this section, we present two advanced techniques that address the above two issues (Sections IV-A and IV-B). Furthermore, such design allows efficient implementation of functions that request memory of an arbitrary size (Section IV-C).

A. A Bitmap of Used Bits

In each step of GETPAGE in the basic RW design, a thread visits one page at a time,. As a result, it could take many steps to find a free page, especially under a low A/T ratio. To remedy that, we use a **Bitmap** to store all pages' *used* bits in consecutive (global) memory space. We can utilize a GPU's high memory bandwidth and in-core computing power to achieve extremely efficient scanning of the bitmap to locate free pages. For example, the Titan V has global memory bandwidth of 650+GBps, and 3072-bit memory bus. Meanwhile, the CUDA API provides a rich set of hardware-supported bit-operating functions. In practice, the bitmap can be implemented as an array of 32-bit or 64-bit integers (words) so that we can visit a group of 32 or 64 pages in a single read. Finding a free page now reduces to finding a word from the bitmap that has at least one unset bit. Such an algorithm (named RW-BM) can be easily implemented by slightly modifying Algorithm 1, as presented in Algorithm 2. The main benefit of RW-BM is much better performance over RW, as shown in the following analysis.

Algorithm 2: GETPAGE based on RW-BM

```
output: the ID of a free page
1: while True do
2:     p ← random integer within [0, T/w)
3:     r ← Atomically set LockMap[p] to 1
4:     P ← (r == 1)? 0xffffffff : BitMap[p]
5:     f ← ffs(P)
6:     set f^th bit in BitMap[p] to 1
7:     return p*w+f-1
8: end while
```

1) Performance of RW-BM: When there are A pages available, and we read w bits at a time, the probability of finding a group with at least a free page is $1 - (\frac{T-A}{T})^w$. Therefore, the expected number of steps for the first thread to find a free page is $\frac{1}{1-(\frac{T-A}{T})^w}$. Following the same logic in deriving Eq. (3), TAS becomes:

$$E(X_i) = \frac{1}{N} \sum_{j=0}^{N-1} \frac{1}{1 - (\frac{T-A+j}{T})^w} \quad (5)$$

Similar to the way we derived Eq. (4), we aim to find an upper bound of *WAS*. $E(X_i)$ is upper bounded by $E(X'_i)$ where X'_i follows a Geometric distribution with probability $1 - (\frac{T-A+N}{T})^w$. The upper bound of WAS becomes

$$E(Y) < \sum_{k=0}^{\infty} \left[1 - \left(1 - \left(\frac{T-A+N}{T}\right)^{wk}\right)^{32}\right] \quad (6)$$

Comparing to Eq. (4), the new bound just added a factor w to the power of the term $\frac{T-A+N}{T}$. The following theorem shows the difference between these two bounds.

Theorem 1. *Denote the upper bound of $E(Y)$ for RW-BM as V', and that for the basic RW algorithm as V, we have*

$$\lim_{A\to N} V' = \frac{V}{w} + \frac{w-1}{2w}$$

Proof. According to the Euler-Maclaurin formula [2]:

$$V' = \sum_{k=0}^{\infty} f(k) = \sum_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{wk}\right)^{32}\right]$$
$$\approx \int_{k=0}^{\infty} f(k)dk + \frac{f(0)+f(\infty)}{2}$$

As $N \to A$, $f(0) \to 1$ and $f(\infty) \to 0$. Therefore,

$$\lim_{A\to N} V' = \frac{1}{2} + \int_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{wk}\right)^{32}\right]dk$$
$$= \frac{1}{2} + \frac{1}{w}\int_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{wk}\right)^{32}\right]dwk$$
$$= \frac{1}{2} + \frac{1}{w}\int_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{k}\right)^{32}\right]$$
$$= \frac{1}{2} + \frac{1}{w}\sum_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{k}\right)^{32} - \frac{1}{2}\right]$$

$$= \frac{w-1}{2w} + \frac{V}{w}$$

Theorem 2. *Denote the TAS $E(x)$ for RW-BM as U', and that for the basic RW algorithm as U, we have*

$$\lim_{A\to N} U' = \frac{U}{w} + \frac{w-1}{2w}$$

Proof. Proof is similar to the proof of Theorem 1.

The above theorems are encouraging in that TAS and the WAS bound decreases by a factor up to w, i.e., 32/64 times as small if we read a 32/64-bit word at a time from the bitmap. More important, the advantage of RW-BM is the highest when AN, which is an extreme case of low free page availability.

For each word in the *used* bitmap, we introduce a *lock* bit and store in another bitmap called **LockMap**. This LockMap is for the implementation of low-cost locks (Section IV-B).

RW-BM is memory efficient: a one-bit overhead is negligible even for page sizes as small as tens of bytes, and the total size of the LockMap is even smaller.

B. Collaborative Random Walk Algorithm

As mentioned earlier, the basic RW design suffers from the large difference in the number of steps for threads in a warp to locate a free page. Our idea to remedy that is to have the threads in the same warp work cooperatively – threads that found multiple pages from the bitmap will share the pages to others that did not find anything. This can effectively reduce the longest steps of RW by the threads. The algorithm runs at two steps: (1) the threads work together to find enough free pages to serve all GETPAGE requests of the entire warp; (2) the identified free pages are assigned to individual threads according to their needs. All threads terminate at the end of step (2), thus divergence is largely eliminated. As a result, we will have the same TAS and WAS values.

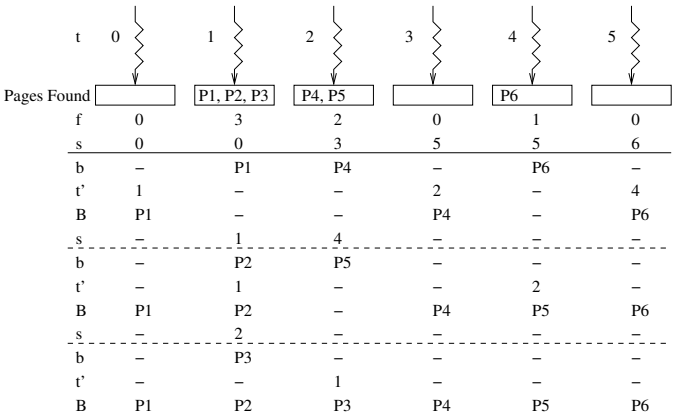

Fig. 7. Step-by-step (from top to bottom) changes of values of key variables in 6 threads running the CoRW algorithm

Efficient implementation of the above idea is non-trivial. The main challenge comes from the needs to keep track of the found pages and distribute them to requesting threads in a parallel way. The SIMD nature of warp execution also requires minimization of code divergence. We design a Collaborative Random Walk (CoRW) algorithm by taking advantage of CUDA *shuffle instructions* that allow access of data stored in registers by all threads in a warp. The design of CoRW is sketched in Algorithm 3. Note that we use many CUDA intrinsic function names in the pseudocode to highlight implementation details, and we will explain what they compute in the following text.

We first get a 32-bit string containing all threads that participate in this process (line 2), from which we also obtain the total number of pages requested (i.e., total number of active threads) in this warp (line 3). After that we start finding and allocating pages. First, each thread reads a random word of the bitmap (line 6) denoted as BitMap[p], of which the number of unset bits (free pages) is stored in a local variable f (line 9). Note the use of LockMap here: we first try to set the value of LockMap[p] to 1, this essentially locks the word BitMap[p] and is done via a single atomic operation (line 7). A key innovation here is: if the word was already locked by other threads (when $r = 1$), we cannot use the word as a source of free pages. Instead of idling, it will return a word with all bits set, and continue the rest of the loop body acting as a consumer of free pages.

We then conduct an exclusive prefix summation of all f values and store the results in s (line 10). Finally, each thread will assign the pages it found to other threads one by one, starting from position (thread ID) s. Specifically, we use a variable b to hold the current page to be assigned (line 12). Following that, we need to ship b to the target thread s. This is tricky in that the CUDA shuffle instructions only allow a thread to copy data from another thread. Our solution is to pass the contributor's lane ID to the receiver's t' variable (line 13). Finally, the page ID in b can be copied and stored in B (line 14). Figure 7 shows an illustrative example, in which we have 6 threads requesting pages. Since threads 1, 2 and 4 found 6 pages altogether, we can serve all requests in one round. Without CoRW, threads 0, 3, 5 will have to access the

bitmap again.

Our CoRW implementation is efficient because all data (other than Bitmap[p]) are defined as local variables and thus stored in registers. Furthermore, all steps (except reading BitMap[p]) are done via hardware-supported functions with extremely low latency. For example, finding the number of set bits (*popc*) in a word can be done in 2 clock cycles, and finding the first unset bit (*ffs*) in 4 cycles. Such latency is in sharp contrast to reading the bitmap from the global memory, which requires a few hundred cycles [4], [19].

Algorithm 3: Collaborative GETPAGE within a warp

input : w: word length, typically 32 or 64
output: ID of a free page acquired

```
t ← lane ID (0-31) of this thread
m ← activemask()
totalNeeds ← popc(m)
totalFound ← 0
while totalFound < totalNeeds do
    p ← random integer within [0, T/w)
    r ← Atomically set LockMap[p] to 1
    P ← (r == 1)? 0xffffffff : BitMap[p]
    f ← w − popc(P)
    s ← exclusive prefix sum of all the f values
    for i : 0 → max(f) − 1 do
        b ← ID of the i-th page found by this thread
        t' ← popc(ballot_sync(s == t))
        B ← shfl_sync(b, t')
        s ← s + 1
    end for
    set corresponding bits in BitMap[p]
    if r == 0 then
        LockMap[p] ← 0
    end if
    totalFound += reduce_add_sync(i)
end while
return B
```

C. Finding Multiple Consecutive Pages

An important extension is to request consecutive memory of an arbitrary size, much like the MALLOC function in C. This is very useful for applications in which threads obtained its memory consumption after the kernel is launched. In fact, our work on RW-BM and CoRW paved the way towards such a procedure (which we name **RW_malloc**). We still divide the memory pool into small units of the same size and store the *used* bits in a bitmap (we will discuss the choice of unit size later). Thus, the problem of getting X bytes by RW_malloc reduces to getting $n = \lceil X/S \rceil$ consecutive units where S is the unit size. Following the RW design, threads scan the bitmap in a parallel and random manner. Instead of a single unset bit, we need to find n consecutive unset bits.

Our design of **RW_malloc** follows the idea of CoRW. However, instead of each thread's reading a random word, all (active) threads in a warp will read in consecutive words to form a large region (e.g., 2048 bits) of the bitmap. Then each thread will scan a small part (e.g., one word) of the region to find all consecutive unset bits (called free segments) in it. Free segments running across two neighboring words will also be connected. Critical information (e.g., starting position, length, used or not) of all free segments are stored in a data structure K. Finally, each thread will traverse K to find a free segment that can serve its RW_malloc request. A sketch of RW_malloc design is shown in Algorithm 4.

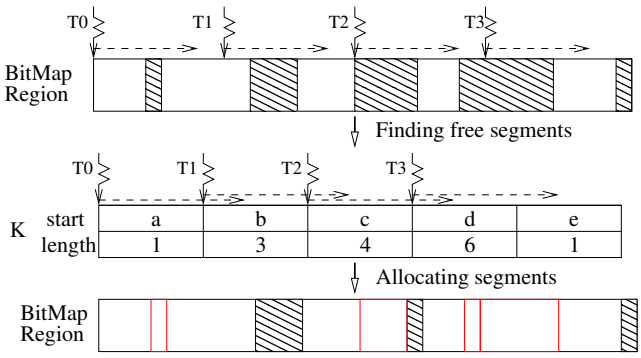

Fig. 8. An example of 4 threads running the RW_malloc algorithm

Due to the fast scanning of the bitmap, RW_malloc inherits the good performance of RW-BM, as multiple bits can be visited at once. As compared to CoRW, the cost of RW_malloc is higher, as we cannot run all computation using intrinsic and shuffle functions. However, since the main data K is stored in shared memory, the overall performance is still orders of magnitude higher than Ouroboros (Section V-B2).

In malloc-style allocations, in addition to latency, we also need to consider utilization of memory. The size of the basic memory unit, the page, is a key parameter that affects space efficiency. As large pages may contain wasted space, we prefer small page sizes. However, When S is too small, we face the challenge of scanning large chunks of the bitmap, leading to degraded RW_malloc performance. Our solution is to aggregate requests for small sizes within a warp and treat the aggregated sizes as a single request. Once we have found consecutive pages that fit the aggregated request, the allocated space is then distributed to the original small request.

Limitation: Our RW_malloc design practically set a cap on the number of consecutive pages: n should be smaller than the size of a bitmap region (e.g., 2048). Support of larger allocations will be an interesting direction for future work. However, we believe the current work has its value. With a large number of threads on GPUs, the requested memory size from each thread tends to be smaller than that in CPU systems. Plus, the range of memory sizes supported in RW_malloc surpasses that by Ouroboros, allowing a meaningful comparison to the best current solution.

D. Clustered Random Walk

The basic setup provides a framework for us to implement more advanced algorithms. To alleviate the fragmentation problem and exploit the spatial property of the Bitmap, we introduce a variation of RW called Clustered Random Walk

Algorithm 4: RW_MALLOC

input : n, number of consecutive pages to find
input : w: word length, typically 32 or 64
output: the ID of the first free page

```
1:  initialize array K in shared memory
2:  t ← lane ID (0-31) of this thread
3:  initialize totalNeeds and totalFound
4:  while totalFound < totalNeeds do
5:    if t == 0 then
6:      p ← random integer within [0, T/w)
7:    end if
8:    P ← BitMap[p+t]
9:    scan P to find all consecutive unset bits
10:   concatenate last unset bit segment to the first one of
      BitMap[p+t+1]
11:   deposit all segments to K
12:   synchronize threads in warp
13:   for i : 0 → length of K do
14:     if K[i].length >= n AND K[i].used == false then
15:       Atomically set the bits of K[i] in BitMap
16:       if Above is a success then
17:         increment totalFound by 1
18:         B ← K[i].start
19:       end if
20:     end if
21:   end for
22: end while
23: return B
```

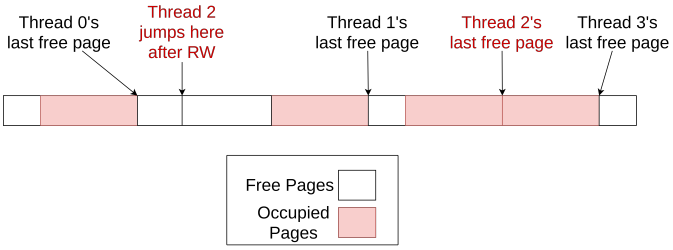

Fig. 9. Illustration of the CRW idea

(CRW). Intuitively, CRW is designed to distribute the free pages into clusters of consecutive pages so that, if one page was found free, its adjacent page(s) are likely to be free. A visual demonstration is presented in Figure 9: all threads keep drawing from their own free-page cluster. Threads 0, 1, and 3 still have free pages in their clusters, so they can quickly grab one. Thread 2 runs out of free pages in its cluster and has to perform a random walk to find a new cluster.

CRW using Bitmap is presented in Algorithm 5. We introduce a thread-level local variable $last\_free\_page$, which stores the ID of the last page obtained by that thread. With that, the CRW algorithm simply stores the last page that each thread has obtained and tries to return the adjacent page to serve the next GETPAGE request. If the adjacent page is not available, CRW calls the regular RW procedure to get a page. Before returning a free page, we need to save the ID of the newly-acquired page to $last\_free\_page$.

CRW has two advantages over RW. First, fragmentation is reduced because used pages are clustered and free pages are also clustered. Second, with a certain (high) probability we can quickly get a page from $last\_free\_page$, thus saving the time to continue the random walk, which is more expensive than accessing $last\_free\_page$.

Algorithm 5: GETPAGE based on CRW

output: ID of a free page acquired

```
1:  p = last_free_page + 1
2:  i = p/32
3:  j = p - i*32
4:  if bit j on Bitmap[i] is 0 then
5:    try to flip it to 1
6:    if above is successful then
7:      last_free_page = p
8:      return p
9:    end if
10: end if
11: p = getPageRandomWalk()
12: last_free_page = p
13: return p
```

1) Performance Analysis of CRW: During the lifetime of a program that utilizes the CRW algorithm, acquired pages tend to occupy consecutive space in the buffer pool. Therefore, at any point, the buffer pool is divided into clusters of consecutive occupied pages and consecutive free pages. This trend may be broken with external fragmentation, i.e., irregular page-freeing patterns that fragment a large cluster into many smaller clusters. Analysis of CRW performance will be based on study of spatial distribution of such clusters.

CRW can be implemented together with the Bitmap data structure. However, to simplify the discussions in this section, we ignore the effects on Bitmap on both CRW and RW. In practice, the benefits of Bitmap are the same to both RW and CRW.

2) Per-Thread Average Steps: Let X'_i be the random variable that represents the number of steps taken until finding a free page by using the CRW algorithm and X_i be that of the RW algorithm. We can represent X'_i with X_i as:

$$X'_i = \begin{cases} 1 \text{ with probability } 1-p \\ 1+X_i \text{ with probability } p \end{cases}$$

The two cases shown above represent the two branches in Algorithm 5, lines 2-8 and lines 9-11, and p is the probability that the page adjacent to the last free page is also free (i.e., branching into lines 9-11). It is easy to see that any moment of X'_i is upper bounded by that of X_i. For example, the first moment of X'_i is:

$$E(X'_i) = 1 - p + p(E(X_i) + 1) = 1 + p * E(X_i)$$

according to the law of total expectation, and we have $E(X'_i) \leq E(X_i) + 1$. The equality is reached when $p = 1$. Therefore, the key parameter for the analysis of $E(X'_i)$ is p.

To formulate p, we need to introduce the concept of free-page clusters. In a buffer pool of T pages, a cluster $[a, b]$ is a set of consecutive free pages, page $a-1$ is occupied (or $a = 0$), and page $b+1$ is also occupied (or $b = T-1$). Since the algorithm tries to get consecutive pages by saving the last free page that it obtained, a thread would keep drawing from one cluster until that cluster is depleted, after which it performs Random Walk to find a new cluster. Therefore, quantity p is the same as the probability that a cluster is depleted in the previous GETPAGE request, that is when $a = b$. We need to mathematically characterize the system at two points in time, the previous GETPAGE request and the current GETPAGE request.

Let M_{t-1} be the number of free-page clusters and $a_{1,t-1}$, $a_{2,t-1}$, ..., $a_{M_{t-1},t-1}$ be the (positive) sizes of the clusters before the GETPAGE request at time $t-1$. The sum of M_{t-1} cluster sizes must be A_{t-1}, which is the total number of free pages in the buffer pool at that moment:

$$a_{1,t-1} + a_{2,t-1} + ... + a_{M_{t-1},t-1} = A_{t-1}$$

Similarly, before the current request:

$$a_{1,t} + a_{2,t} + ... + a_{M_{t-1},t} = A_t \tag{7}$$

Note that in this , $a_{i,t}$ can be 0 and we do not update the time index for M_{t-1}. The probability $p = P(a_{i,t} = 0)$ can be calculated as follows.

$$p = \frac{\text{number of solutions to Eq. (7) where } a_{i,t} = 0}{\text{number of solutions to Eq. (7)}}$$

Eq. (7) is a simple linear Diophantine and its number of solutions has been proven by using the stars-and-bars representation [45] as follows. Suppose that there are A_t stars and $M_{t-1} - 1$ bars, an arrangement of the stars and bars is equivalent to one solution of Eq. (7) where the number of stars between two bars equals the value of one $a_{i,t}$. For example, let $A_t = 10$ be the number of stars and $M_{t-1} - 1 = 3$ be the number of bars. One possible arrangement of 10 stars and 3 bars is

$$*|**||*******$$

This arrangement is equivalent to the solution $a_{1,t} = 1$, $a_{2,t} = 2$, $a_{3,t} = 0$, $a_{4,t} = 7$ for Eq. (7) where $A_t = 10$ and $M_{t-1} = 4$. Note that there is no star between the second and third bar, which is equivalent to $a_{3,t} = 0$.

Following this logic, the total number of solutions to Eq. (7) is the number of arrangements of A_t stars and $M_{t-1} - 1$ bars, which is the number of ways to select $M_{t-1}-1$ positions among $A_t + M_{t-1} - 1$ positions to insert the bars (and thus leave the remaining A_t position for the stars). The number of ways to select $M_{t-1} - 1$ positions among $A_t + M_{t-1} - 1$ positions is:

$$\binom{A_t}{M_{t-1}} = \frac{(A_t + M_{t-1} - 1)!}{(M_{t-1} - 1)!A_t!}$$

This is the total number of solutions to Eq. (7). Now we find the number of solutions to Eq. (7) where one $a_i = 0$. Given one $a_i = 0$, Eq. (7) becomes

$$a_{1,t} + a_{2,t} + ... + a_{M_{t-1}-1,t} = A_t$$

Similar to the above, the total number of solutions to this equation is

$$\binom{A_t}{M_{t-1} - 1} = \frac{(A_t + M_{t-1} - 2)!}{(M_{t-1} - 2)!A_t!}$$

Therefore, we have

$$\begin{aligned} p &= P(a_{i,t} = 0) \\ &= \frac{(A_t + M_{t-1} - 2)!(M_{t-1} - 1)!A_t!}{(M_{t-1} - 2)!A_t!(A_t + M_{t-1} - 1)!} \\ &= \frac{M_{t-1} - 1}{A_t + M_{t-1} - 1} \end{aligned}$$

To simplify, we drop the time index:

$$p = \frac{M - 1}{A + M - 1} \tag{8}$$

where A is the current number free pages and M is the number of clusters before the previous GETPAGE request.

Following that, TAS is:

$$\begin{aligned} E(X_i') &= 1 - p + p(E(X_i) + 1) \\ &= 1 + \frac{M - 1}{A + M - 1} \times \frac{T}{N} ln\left(\frac{A}{A - N}\right) \end{aligned} \tag{9}$$

When there are few large clusters, $M \to 1$ and $E(X_i') \to 1$. When there are many small clusters, $M \to T$ and $E(X_i')$ approaches the $E(X_i)$ value of the RW algorithm. Therefore, under a circumstance where an irregular page-freeing pattern breaks the space into many small clusters, we will see that the performance of CRW converges to that of RW. This shows that CRW is of significant intellectual and practical value as it will always outperform RW.

In Figure 10, we plot the theoretical TAS values of the RW (Eq. (3)) and CRW (Eq. (9)) algorithms under four scenarios with different percentage of free pages. For each scenario, we show the results of CRW using three different M values. In all such cases, the TAS value of CRW is much smaller than that of RW. Even under the situation of $M = A$ (CRW3) and 0.5% free pages, RW is about twice as large as RW.

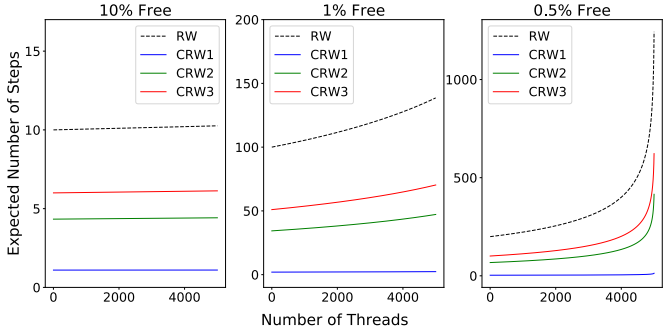

Fig. 10. Expected number of steps of CRW according to Eq. (9) under different M values (CRW1: M = 0.01A, CRW2: M = 0.5A, CRW3: M = A) and percentage of free pages in comparison to RW

3) Per-Warp Average Steps: Now we develop analysis of CRW with the second metrics. The intra-warp max is $Y_i' = max(X_0', X_1', ..., X_{31}')$. Since some X_i' are 1 and other $X_i' = X_i$, Y_i' is the maximum of a number Q of random variable X_i where $Q \leq 32$:

$$Y_i' = max(X_0, X_1, ..., X_Q)$$

The expectation of Y_i', $E(Y_i')$, should be upper-bounded by WAS of RW, $E(Y_i)$, because Y_i is the maximum of 32 random variables X_i. Therefore, $E(Y_i')$ is also upper bounded by $E(Y_i)$'s upper bound, which is presented in (4). This means that, with respect to the intra-warp max. steps, CRW is (upper) bounded by RW. A closed-form estimate of $E(Y_i')$ is very difficult to achieve analytically because Q itself is also a random variable.

V. Experimental Evaluation

A. Experimental Setup

In this section, we conduct experiments to evaluate the performance of the RW-base algorithms. We perform four sets of experiments to compare our methods with Ouroboros in a unit-test setup. In all experiments, we configure both Ouroboros and our systems to have a total of 10 GB in memory pool and to serve the maximum request size of 8192B. Each chunk in the Ouroboros system is 8192B large and there are ten processing queues which process requests of size 8192B, 4096B, 2048B, etc. We use the same Ouroboros code and environment configurations as presented in [57] to ensure a fair and meaningful comparison. On our side, we evaluate three types of systems: basic Random Walk without bitmap (RW), Random Walk with Bitmap (RW-BM), and Collaborative Random Walk (CoRW). In RW-BM and CoRW, we use 32-bit words for the Bitmap. We built all our code under CUDA 11.4 and Linux Ubuntu 18.04 version. We run all experiments in a workstation with an AMD Ryzen Threadripper 1950X CPU, 128GB of DDR4 memory, and an NVidia Titan V GPU. Unless specified otherwise, each data point presented in all figures is the average of 100 experimental runs with the same parameters.

Metrics: First, we measure the total running time of the kernels in all experiments. In conformity with our analytical work, our experiments also evaluate the following two metrics: Per-Thread Average Steps (TAS) and Per-Warp Average Steps (WAS). Since Ouroboros does not allow extraction of TAS and WAS without a structural change, we implemented a simple queue-based program to collect TAS and WAS data. We implemented the queue on an array with a pointer to the first element of the array. Threads need to use an atomic operation to shift the first element pointer to obtain a page. The number of steps in this implementation is measured based on the position of the obtained page. For example, if a thread obtains the fourth free page, we determine that it takes 4 steps to obtain that page. The first three steps are stalls for some other three threads to obtain the first three pages, and one step for the mentioned thread to obtain the fourth page. The number of steps in RW and CoRW kernels is measured by the number of pages a thread has visited until it successfully locks in a free page. TAS is measured by taking the average of all step counts recorded. To calculate WAS, we first find the maximum number of steps within each warp and then take the average of these quantities among all warps.

B. Experimental Results

1) Performance of GetPage*:* First, we evaluate the performance of the four programs in getting a single page. Specifically, we develop a single GPU kernel whose only task is to request a page from the memory buffer pool. The equivalent in the Ouroboros system is a kernel that requests 256B for each thread because 256B is one of the page units. We launch the kernel with various numbers of threads (i.e., changing the N value) and various numbers of free-page percentage (i.e., changing the A value). The buffer pool has a total of T=1M pages unless specified otherwise.

Figure 11 shows the three metrics measured from kernels that call Ouroboros (simple queue in case of TAS and WAS), RW, RW-BM, and CoRW. The simple queue was implemented as described in subsection V-A. The four columns represent scenarios with different free-page percentages. In each scenario, we pre-set some pages as occupied so that the percentage of free pages before starting the kernels is 50%, 10%, 1%, and 0.5%, respectively. Results from the first row shows that our most advanced method CoRW outperforms Ouroboros by more than an order of magnitude under most of the cases. When there are less than 1% free pages, performance of our methods start decreasing, but the CoRW still outperforms Ouroboros by a big margin. Note that the 1% is a really extreme case that is not expected to happen frequently in applications – it means that after serving all the requests, there are 0 pages left. Another observation is: the Ouroboros running time increases with number of threads but our algorithm is insensitive to that (except for the 0.5% free page case).

Results from the second and the third rows confirm the validity of our theoretical results (i.e., Equations (3), (4), (5), and (6)) First, the measured TAS values match the theoretical results well. The theoretical upper bound of WAS matches experimental results well even under 1% of free pages, indicating the bounds are tight. The bound becomes loose as the percentage of free page decreases to below 1% . Another observation is that WAS is a better indication of the total running time than TAS. Visually, the growth patterns of lines in the 1st row of Figure 11 matches better with that on the third row (WAS) than on the second row (TAS).

Figure 12 shows the ratio of RW over RW-BM in terms of TAS and WAS. The smooth lines on the TAS plot is the ratio of the numerical values of Eq. (3) over those of Eq. (5). Those on the WAS plot are the ratio of the numerical values of Eq. (4) over those of Eq. (6). As the free-page percentage decreases, the ratio becomes bigger. When the percentage approaches zero, both the theoretical ratios and the measured ratios are very close to 32. This validates Theorem 1.

2) Performance of RW_malloc: Second, we evaluate the performance of our approach in allocating memory with various sizes. Seven scenarios are tested: each thread requests 16B, 256B, 1024B, 4096B, 8192B, mix request sizes ranging from

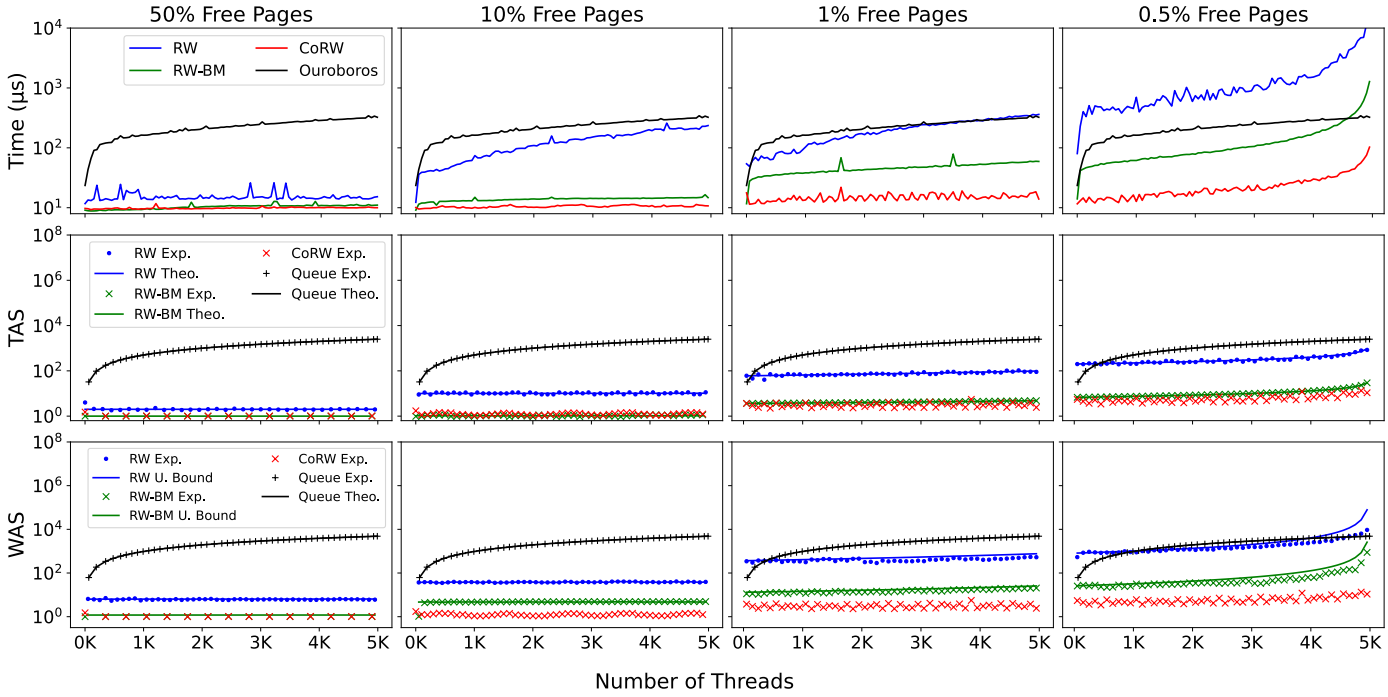

Fig. 11. Performance of our unit-test kernel calling GETPAGE under different numbers of parallel threads and percentage of free pages. Displayed free percentages are measured at the start of each kernel.

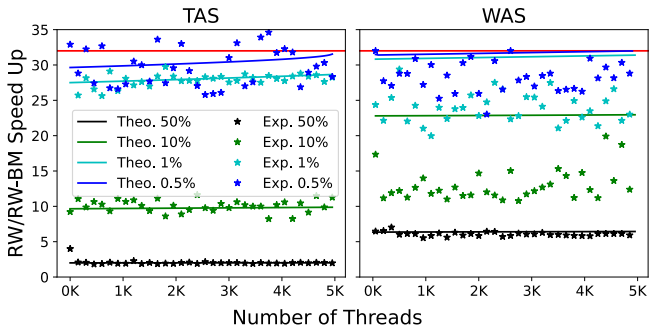

Fig. 12. Speedup of RW-BM over RW. Theoretical results are smooth lines calculated by Eq. 3 and 5 for TAS and Eq. 4 and 6 for WAS.

4B to 8196B, and 4096B but immediately frees the allocated space. The last scenario represents the same setting presented in the survey paper [59]. In each scenario, we launch the kernel with various numbers of threads and free-page percentage. Since the largest request size that we need to support is 8192B, we choose the page size of 256B (8192/32) to fit this maximum size in the single word of the Bitmap.

Figure 13 presents the total kernel time of RW_malloc and Ouroboros in the seven scenarios. This figure shows that RW_malloc outperforms Ouroboros in allocating a wide range of memory sizes. The improvement is up to 2 orders of magnitude in the best cases where there are the most number of parallel threads and most free memory units in the system. The higher the concurrency is, the better RW_malloc performs than Ouroboros due to Ouroboros' linear scaling. RW_malloc's performance degrades as fewer memory units are available. However, it is still much better than Ouroboros in its worst scenario when there are almost no free pages (note that logarithmic scale of Y-axis). The only case where Ouroboros wins is when it immediately frees memory just allocated. By this, Ouroboros hits a sweet spot - it does not need to allocate new chunks nor extend the virtualized queues. However, this is definitely an unrealistic scenario, as buffers will normally be used before released. The difference between RW_malloc and Ouroboros in mixed-size allocation is consistent throughout all size ranges, as demonstrated in Figure 14.

For RW_malloc, we also evaluated memory utilization. In this experiment, we keep sending memory allocation requests until a system reports an out-of-memory error. The memory utilization rate is then calculated as the fraction between the total allocated amount and the total amount of memory in the system. This design is similar to the Out-Of-Memory testcases in the survey paper [59]. We perform this experiment with various unit sizes while maintaining the fairness between Ouroboros and our system in terms of total memory and maximum allocation size. For example, an Ouroboros' chunk size of 4096B is equivalent to the page size of 128B in our system because both systems can support a maximum allocation size of 4096B. Figure 15 presents the results of the memory efficiency experiment. Both systems achieve very good memory utilization rate when allocation sizes are some power of two because these sizes fit perfectly into some number of pages. This finding is consistent withthe Out-of-Memory testcases in [59]. However, the CoRW/RW-BM system significantly outperforms in other situations. The reason is that our system allocates large memory chunks by aggregating

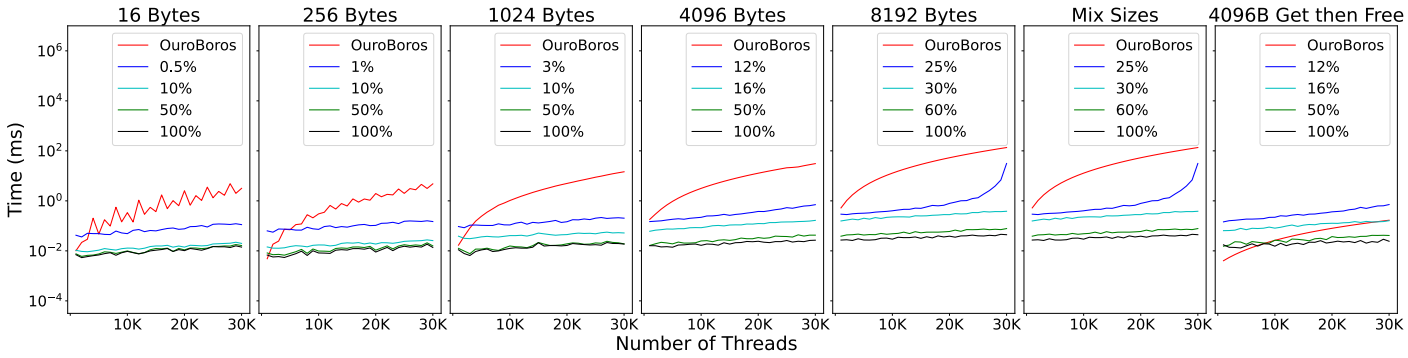

Fig. 13. Performance comparison between RW_malloc and Ouroboros in allocating multiple pages at a time. From left to right: (1) allocate 16 bytes, (2) allocate 256 bytes, (3) allocate 1024 bytes, (4) allocate 4096 bytes, (5) allocate 8192 bytes, (6) allocate mix sizes, (7) allocate 4096 bytes then free immediately. Displayed free percentages are measured at the start of each kernel.

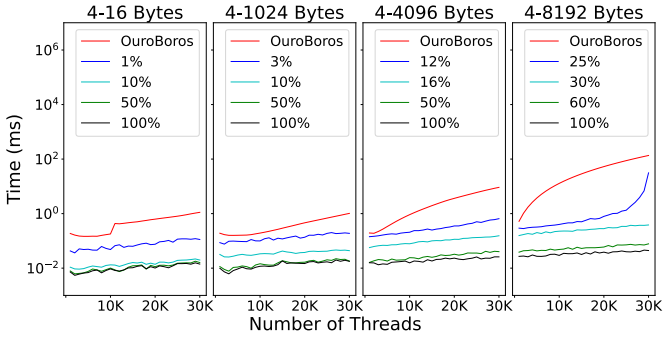

Fig. 14. Performance comparison between RW_malloc and Ouroboros in allocating mixed sizes.

small consecutive pages and thus has a finer granularity control over the memory space. For example, to allocate 1050B, our system rounds the size to 1280 (256*5) and allocates five consecutive pages whereas Ouroboros has to round to 2048B and allocates one page of that size.

VI. Case Study

In this section, we report experimental results of GPU programs with page acquisition needs served by our parallel memory management APIs. As compared to unit-tests discussed in Section V, this gives us a approach to evaluate our methods in real-world applications. In particular, we focus on the join database operator implemented as part of a GPU-based DBMS, in which a global dynamic memory pool is maintained and shared by all GPU processes. The experimental environment is the same as we described in Section V.

Join is arguably the most important relational operator. Among the several types of join algorithms, we use a state-of-art hash join designed for GPUs [46] as the foundation of this case study. Based on the popular idea of radix hashing, the algorithm includes three stages: partitioning input data, building a hash table, and probing. We focus our discussions on the probing stage because that is where our buffer management APIs are needed (to allocate memory for output data). After building histograms for hash values and reordering both input tables in the first two stages, the probing will compare tuples of the corresponding partitions of the two tables and output one tuple whenever there is a match (Figure 16). We modified the code from [46] such that, when a thread finds a match, it will write the results to the current data page; when the page is full, the thread will ask for another page by calling GETPAGE.

We measured the end-to-end processing time of all stages of the hash join code augmented by various GETPAGE implementations: Ouroboros, RW, RW-BM, and CoRW. We also include the original code used in [46] named Direct Output Buffer (DO). The DO design assumes pages are never freed therefore GETPAGE is done by simply incrementing a global counter via atomic operations. DO is shown [46] to outperform the double-computation solution by more than 25%.

We first run the code under different input table sizes from 16K to 10M tuples while fixing the total page number to 128M. By that, we achieve smaller percentage of free pages with the increase of the data (table) size. Note that the data size is roughly equal to the total number of threads.

According to Figure 17, by using Ouroboros, the join kernel runs slower than others by a large margin.

The results show DO is slightly better than RW since it only involves an *atomicAdd* operation. With the help of bitmap implementation, our RW-BM and CoRW alogorithms outperform DO in all cases and CoRW is the best among them all.

In the second setup, we test different page sizes with different algorithms. In order to explore the possibility of different page size, we need a large enough buffer pool that guarantees that the smallest number of pages (the largest page size) can fulfill the total number of page requests. The data size is fixed to 4M, and we allocate a total of 8GB space for the buffer pool with all pages are free, the page size varies from 32 bytes to 2KB. The results of choosing different page sizes from 32 bytes to 2KB are shown in Figure 18. We do not plot Ouroboros in time figure but we use it as the baseline to show speedups of other algorithms.

Observing the results, we can see that Do is slightly better than RW, RW-BM and CoRW outperforms DO, and CoRW is the winner in all cases. We also notice that the performance change of all the algorithms follow the same trend: the running time decreases when page size increasing. But the reasons are different. The processing time of DO decreases is because as

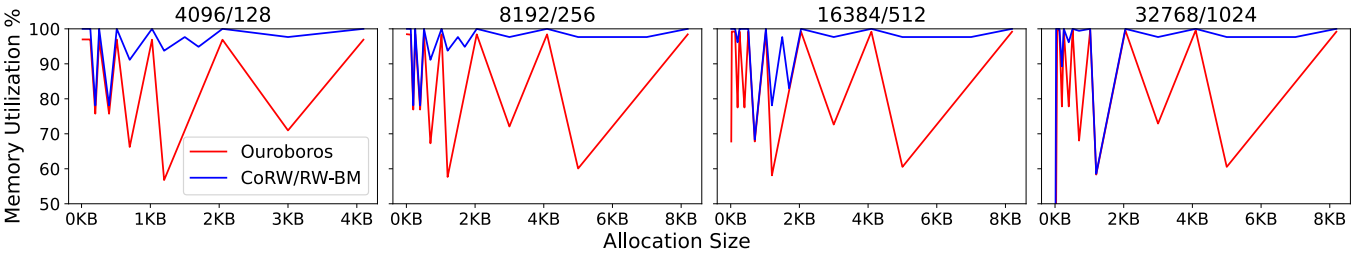

Fig. 15. Memory efficiency in four pairs of system settings: (1) Ouroboros 4096B chunk versus CoRW 128B page, (2) Ouroboros 8192B chunk versus CoRW 256B page, (3) Ouroboros 16384B chunk versus CoRW 512B page, and (4) Ouroboros 32768B chunk versus CoRW 1024B page

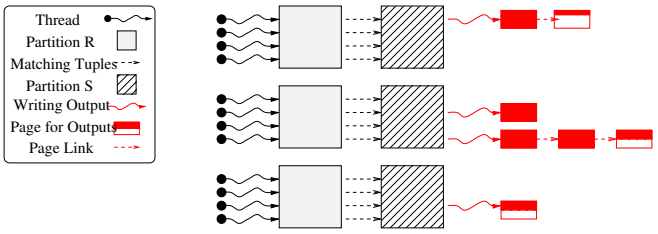

Fig. 16. The Probing stage of GPU hash join

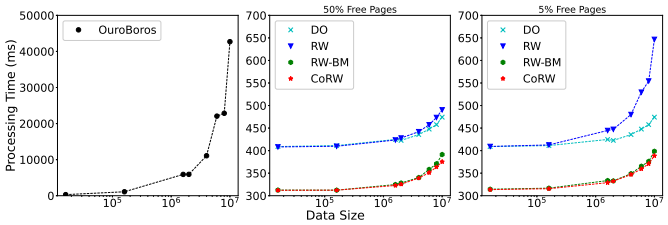

Fig. 17. Total running time of a GPU hash join program enhanced with different buffer management mechanisms under different input table sizes

the page size increases, each thread will have more sufficient space to output results, it will reduce the number of atomic requests to get a page. While for RW-BM and CoRW, since we measured the end-to-end time including memory allocation and transferring time in the case study, the time and space of using to allocate page maps (only related to total number of pages) decreases that leads to the decline.

In summary, our page allocation mechanisms significantly improved the state-of-art join code on GPUs. In comparison, Ouroboros is not suitable for such use cases.

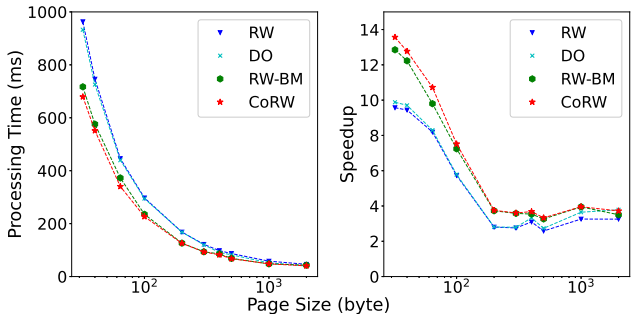

Fig. 18. Total processing time (left) and speedup over Ouroboros (right) of a GPU hash join program enhanced with RW, RW-BM, and CoRW under different page sizes.

VII. Related Work

Memory Management on GPUs: NVIDIA initially announced its dynamic memory allocator for GPUs in 2010. It provides the usual malloc/free interface and can be called by threads in a CUDA kernel. Unfortunately, very little information is known about its internal design [59]. XMalloc [28] was also introduced in 2010 and became the first non-proprietary dynamic memory allocator for GPUs. Its main contribution is the coalescing of allocation requests on the SIMD width for faster queue processing. Allocations are served from a heap that is segmented into blocks and bookeeping information is stored in a linked-list. The linked-list is a major bottleneck because a thread has to traverse through the list of memory blocks when searching for a free one. ScatterAlloc [48] was introduced in 2012 and addressed this bottleneck by scattering the allocation requests across its memory regions. A hash function is used to search for free regions; if the hash value points to an allocated region, linear probing is used to find the nearest free region. FDGMalloc [55] (2014) presents a warp-level optimized approach that aggregates all requests in a warp and chooses a leader thread to traverse through a linked-list of free pages. Adinetz and Pleiter [3] proposed Halloc in 2014; the main idea is to use a deterministic hash function to traverse through memory chunks and to use slab allocation to improve fragmentation. Vinkler and Havran (2015) [53] proposed Register Efficient Memory Allocator (RegEff). RegEff splits the bookeeping information into many linked-lists. During allocation, a thread picks a linked-list and traverse to find the first free chunk that is large enough for the allocation. If the chunk is large enough according to a maximum-fragmentation parameter, it is split into two chunks during the allocation and the linked-list is updated.

Ourobors [57] (2020) is the latest development that introduces the concept of virtualized queues. Similar to a traditional memory allocator, Ouroboros divides its managed memory region into *chunks*. Each chunk contains some metadata and a number of smaller memory regions called *pages*. All *chunks* have the same size and the size is fixed. *Pages* on the same *chunk* also have the same size, but this size may differ across chunks. The Ouroboros system cannot change the page size of a *chunk* when the chunk is in used, but can reassign a different page size to a *chunk* when it is completely free. The largest

page size is equal to the size of the chunk, i.e., the chunk only has one page. Each chunk can have one of the three functionalities: ChunkIndex-chunk, PageIndex-chunk, and Queue-chunk. ChunkIndex-chunks and PageIndex-chunks both hold *pages* that can be allocated to programs; they only differ in the way of managing the *free* operation. Queue-chunks, however, does not hold memory regions that can be allocated to programs; they act as a global state for holding information about free pages/chunks and occupied pages/chunks. To reduce thread contention and increase performance, the Ouroboros system consists of multiple queues and each queue only handles pages of the same size which is twice the page size of the next smaller queue. For example, in a system where chunk size is 8192B and three queues are allowed, the three page sizes that are managed by the three queues are 8192B (largest page size), 4196B, and 2048B. To address the issue of memory overhead for managing the queues, the authors introduced *Virtualized Queues*. The main idea of *Virtualized Queues* is that a new chunk is only allocated to become a Queue-chunk when one of the queues need more space and the physical memory address of an element on a queue can be calculated from the virtualized queue index. In this design of *Virtualized Queues*, the authors avoided pre-allocating memory for all queues, which has a large memory overhead because every queue needs to be sufficiently large to be able to hold all the page indices in the worst case. In a 2021 comparison study [59], Ouroboros showed better performance and better fragmentation than all the aforementioned allocators. Therefore, it can be considered the state-of-the-art.

Memory Management on OS/DBMS: Memory management on traditional (single-thread or low-concurrency) CPU-based systems, let it be OS or DBMS, has been thoroughly studied. On the OS side, Kilburn *et al.* [29] brought up the idea of paging in Atlas system. Page sgementation was firstly discussed by Dennisó *et al.* [15]. Then Corbató *et al.* [14] supported page segmentation in their MULTICS system, which is the first one who achieved that. On the DBMS side, early work can be traced back to Stonebraker [49] that discussed OS supports in the context of a DBMS. Effelsberg *et al.* [17] discussed database buffer manager as a component of DBMS and implemented it. Chou *et al.* [13] presented a DBMIN algorithm to manage the buffer pool of an RDBMS. Chen *et al.* [12] proposed a query execution feedback model to improve DBMS buffer management. Brown *et al.* [9] introduced the concept *hit rate concavity* and developed a goal-oriented buffer allocation algorithm called Class Fencing.

Database Systems on GPUs: Besides individual database tasks, much efforts are dedicated to database system software design on GPUs. A queueing system that schedules and merges CUDA kernels within one kernel to achieve task parallelism was proposed by Guevara *et al.* [24]. To execute a given query plan tree in post-order sequence, Yuan *et al.* [62] developed a query engine that adopts a block-oriented execution model on GPUs. Zhang *et al.* [63] proposed a kernel-adapter GPU-based DBMS called OmniDB using a hardware oblivious database kernel (qkernel) to maximize the common functionality. To support concurrent query processing on GPUs, Wu *et al.* [60] developed a compiler and runtime infrastructure called RedFox to execute relational queries. Paul *et al.* [43] implemented a novel pipelined query execution engine called GPL for query co-processing on the GPU. Wang *et al.* [54] proposed MultiQx-GPU to support concurrent query processing by enabling GPU resource sharing among database queries. Li *et al.* [33] developed a two-stage model towards resource allocation to support heterogeneous queries. Quite a few GPU-based DBMSs are developed in the academia [32] and the commercial world [23], [30], [38], [41].

Synchronization Primitives on Parallel Systems: Atomicity and synchronization are well-known issues in a parallel system [42]. At the system level, the implementation of synchronization primitives were discussed in [22], [25], [37] and a buffer framework was implemented in [16]. On the DBMS side, Barve *et al.* [7] provides a framework for online prefetching and buffer management algorithms in parallel I/O systems. However, none of them has dealt with the parallelism level at the order of magnitude in thousands or tens of thousands, which are normal on GPUs. Xiao *et al.* [61] introduce an inter-block GPU communication via barrier synchronization. Stuart *et al.* [50] revisited the design of synchronization primitives and applied them to the GPU, including Spinlock and FA lock.

VIII. Conclusions and Future Work

In this paper, we study the problem of buffer management in highly parallel software systems, with a focus on GPU systems. The main idea of our design relies on random processes to locate free buffer pages and avoids maintaining a centralized data structure that could become a major bottleneck in a high-concurrency systems. Based on that philosophy, we propose a memory page allocation design based on a Random Walk (RW) mechanism. We have proven mathematically that RW can significantly outperform any queue-based solution under the vast majority of scenarios. RW shows its limitations in extreme cases when free buffer pages are very rare. To remedy that, we propose two advanced techniques: the first one (named RW-BM) is based on the storage of page information in a bitmap. This is shown to improve the latency of RW by a factor of 32 or 64. The second one (named CoRW) involves the sharing of free pages among neighboring threads which can further lower the number of steps to find a page. We also present our initial solution to the problem of allocating multiple pages. The results from experimental unit-tests are all consistent with the mathematical analyses. The results show that our solutions outperform the best known GPU memory allocator, Ouroboros, by more than two orders of magnitude. Furthermore, we demonstrate a case study by integrating the page allocation code into a state-of-the-art GPU-based hash join algorithms and showing significant performance boost.

Our idea provides a framework that can be extended to accommodate a wide range of algorithms to gain better performance under different scenarios. More aggressive random walk approaches can be designed and analyzed. The RW_malloc design still has its limitations, i.e., supporting very large memory requests can be a interesting research topic.

REFERENCES

[1] AMD Ryzen™ Threadripper™ 3990X Processor.

[2] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, pages 16,806,886. New York: Dover Publications.

[3] A. V. Adinetz and D. Pleiter. Halloc: a high-throughput dynamic memory allocator for gpgpu architectures. In *GPU Technology Conference (GTC)*, volume 152, 2014.

[4] Y. Arafa, A.-H. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz. Low overhead instruction latency characterization for nvidia gpgpus, 2019.

[5] I. Arefyeva, D. Broneske, G. Campero, M. Pinnecke, and G. Saake. Memory management strategies in cpu/gpu database systems: A survey. In *International Conference: Beyond Databases, Architectures and Structures*, pages 128–142. Springer, 2018.

[6] T. Baroudi, V. Loechner, and R. Seghir. Static versus dynamic memory allocation: a comparison for linear algebra kernels. In *IMPACT 2020, in conjunction with HiPEAC 2020*, 2020.

[7] R. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive parallel disk prefetching and buffer management. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 47–56, 1997.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[9] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. *ACM SIGMOD Record*, 25(2):353–364, 1996.

[10] T. Burić and N. Elezović. Approximants of the euler–mascheroni constant and harmonic numbers. *Applied Mathematics and Computation*, 222:604–611, 2013.

[11] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[12] C. M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. Technical report, 1998.

[13] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.

[14] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[15] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM (JACM)*, 12(4):589–602, 1965.

[16] I. Di Gennaro, A. Pellegrini, and F. Quaglia. Os-based numa optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 291–300. IEEE, 2016.

[17] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, 1984.

[18] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis. Unlocking energy. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 393–406, 2016.

[19] M. Fang, J. Fang, W. Zhang, H. Zhou, J. Liao, and Y. Wang. Benchmarking the gpu memory at the warp level. *Parallel Computing*, 71:23–41, 2018.

[20] I. Gelado and M. Garland. Throughput-oriented gpu memory allocation. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 27–37, 2019.

[21] W. Gloger. Wolfram gloger's malloc homepage, 2006.

[22] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 64–75, 1989.

[23] D. Gray. Sqream technologies-removing limits of sql databases. dataconomy, 2014.

[24] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, volume 9, 2009.

[25] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132, 1991.

[26] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):1–39, 2009.

[27] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[28] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139. IEEE, 2010.

[29] T. Kilburn, D. B. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[30] Kinetica. Maximizing Data Analytics Price/Performance WITH GPU ACCELERATION.

[31] D. E. Knuth. Johann faulhaber and sums of powers. *Mathematics of Computation*, 61(203):277–294, 1993.

[32] H. Li, C. Mou, N. Pitaksirianan, R. Rui, Z. Nouri-Lewis, M. Eslami, R. Sheng, S. Lei, J. Wang, and Y. Tu. Cheetahdb®: A system for high-throughput database processing on gpus.

[33] H. Li, Y.-C. Tu, and B. Zeng. Concurrent query processing in a gpu-based database system. *PloS one*, 14(4):e0214720, 2019.

[34] H. Li, D. Yu, A. Kumar, and Y. Tu. Performance modeling in CUDA streams - A means for high-throughput data processing. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 301–310, Oct 2014.

[35] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[36] C. McGinnis. Pci-sig® fast tracks evolution to 32gt/s with pci express 5.0 architecture. *News Release, June*, 7, 2017.

[37] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, 1997.

[38] T. Mostak. An overview of mapd (massively parallel database). *MIT Technical Report*, 2013.

[39] NVidia. NVIDIA TESLA V100 GPU ARCHITECTURE.

[40] NVidia. CUDA C Programming Guide, March 2018.

[41] A. Ocsa. Sql for gpu data frames in rapids accelerating end-to-end data science workflows using gpus. In *LatinX in AI Research at ICML 2019*, 2019.

[42] D. Padua. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.

[43] J. Paul, J. He, and B. He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.

[44] I. Pletnev, A. Erin, A. McNaught, K. Blinov, D. Tchekhovskoi, and S. Heller. Inchikey collision resistance: an experimental testing. *Journal of cheminformatics*, 4(1):1–9, 2012.

[45] J. J. Quinn and A. T. Benjamin. *Proofs That Really Count: The Art of Combinatorial Proof*. The Mathematical Association of America, 2003.

[46] R. Rui and Y.-C. Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, pages 17:1–17:12, New York, NY, USA, 2017. ACM.

[47] M. Springer and H. Masuhara. Dynasoar: a parallel memory allocator for object-oriented programming on gpus with efficient memory access. *arXiv preprint arXiv:1810.11765*, 2018.

[48] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE, 2012.

[49] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[50] J. A. Stuart and J. D. Owens. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

[51] R. Team et al. Rapids: Collection of libraries for end to end gpu data science, 2018.

[52] TechPowerUp. NVIDIA A100 PCIe, 2021.

[53] M. Vinkler and V. Havran. Register efficient dynamic memory allocator for gpus. In *Computer Graphics Forum*, volume 34, pages 143–154. Wiley Online Library, 2015.

[54] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[55] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th workshop on general purpose processor using graphics processing units*, pages 120–126, 2013.

[56] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, pages 1–116. Springer, 1995.

[57] M. Winter, D. Mlakar, M. Parger, and M. Steinberger. Ouroboros: virtualized queues for dynamic memory management on gpus. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[58] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 754–766. IEEE, 2018.

[59] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–233, 2021.

[60] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 44. ACM, 2014.

[61] S. Xiao and W.-c. Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[62] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[63] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013.

[64] G. Y. Zou. Toward using confidence intervals to compare correlations. *Psychological methods*, 12(4):399, 2007.