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

Minh Pham
University of South Florida
Tampa, FL, USA
minhpham@usf.edu

Hao Li
University of South Florida
Tampa, FL, USA
hao.li@kla-tencor.com

Yongke Yuan
Beijing University of Technology
Beijing, China
yyk@bjut.edu.cn

Chengcheng Mou
University of South Florida
Tampa, FL, USA
chengcheng@usf.edu

Kandethody Ramachandran
University of South Florida
Tampa, FL, USA
ram@usf.edu

Zichen Xu
Jiaxing Neofelis Scientific, Inc.
Nanchang, China
zichenxu@outlook.com

Yicheng Tu
University of South Florida
Tampa, FL, USA
tuy@usf.edu

ABSTRACT

Due to the high level of parallelism, there are unique challenges in developing system software on massively parallel hardware such as GPUs. One such challenge is designing a dynamic memory allocator whose task is to allocate memory chunks to requesting threads at runtime. State-of-the-art GPU memory allocators maintain a global data structure holding metadata to facilitate allocation/deallocation. However, the centralized data structure can easily become a bottleneck in a massively parallel system. In this paper, we present a novel approach for designing dynamic memory allocation without a centralized data structure. The core idea is to let threads follow a random search procedure to locate free pages. Then we further extend to more advanced designs and algorithms that can achieve an order of magnitude improvement over the basic idea. We present mathematical proofs to demonstrate that (1) the basic random search design achieves asymptotically lower latency than the traditional queue-based design and (2) the advanced designs achieve significant improvement over the basic idea. 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.

ACM Reference Format:
Minh Pham, Hao Li, Yongke Yuan, Chengcheng Mou, Kandethody Ramachandran, Zichen Xu, and Yicheng Tu. 2022. Dynamic Memory Management in Massively Parallel Systems: A Case on GPUs. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524059.3532387

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532387

1 INTRODUCTION

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. [25, 33, 39–41] Dynamic memory allocation on GPUs was first introduced about ten years ago by NVIDIA and many other solutions have been proposed since then [38]. Many GPU-based applications benefit from dynamic memory allocation such as graph analytics [9, 37], data analytics [5, 31], and databases [4, 18].

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 [38] 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 [38]. For example, the state-of-the-art solution, Ouroboros, 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 [38]. Nevertheless, we show in section 2 that thread contention, synchronization overhead, and memory overhead are still problematic with such solutions.

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. Existing memory allocators maintain a global state (e.g., head and tail of a queue) to keep track of available memory units. The allocation/deallocation 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. [15, 23] 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.

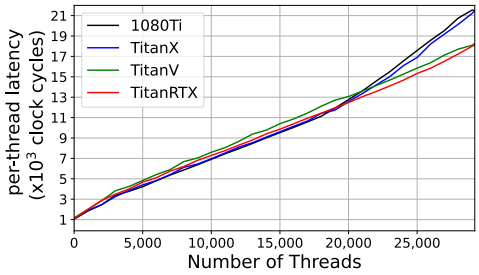


Figure 1: Average latency in accessing a global variable via atomic operations in different NVidia GPUs.

Figure 1 reports our collected data for 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.

In this paper, we present a high-performance memory allocation framework for 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 state explicitly, we let the threads statistically find 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 and a page sharing mechanism among neighboring threads that essentially minimizes resource waste due to code divergence. Based on these two techniques, we also develop an algorithm for serving requests of multiple consecutive pages. The performance advantage of our solutions is 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.

Paper Organization: Section 2 sketches the current state-of-the-art and its drawbacks; Section 3 introduces our memory management framework, key algorithms, and the mathematical reasoning behind our designs; Section 4 presents advanced techniques with improved performance; Section 5 shows results of experimental evaluation (unit-tests and a case study) in comparison with the state-of-the-art; Section 6 presents a brief survey of related work, and Section 7 concludes this paper.

2 BACKGROUND

Dynamic Memory Allocation in CPU-based Systems. Memory allocators on CPUs have been well studied since the 1960s [35]. 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 [17] and the Hoard malloc [6], both of which use multiple arenas for concurrent processing.

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 [19] 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 [24]. That started a series of efforts on this topic, including XMalloc (2010) [20], ScatterAlloc (2012) [29], FDGMalloc (2013) [34], HAlloc (2014) [2], Reg-Eff (2015) [32], DynaSOAr (2019) [28], and Ouroboros (2020) [36]. A recent comparative study [38] showed that Ouroboros outperformed all aforementioned methods in both allocation performance and space efficiency and thus can be considered the state-of-the-art.

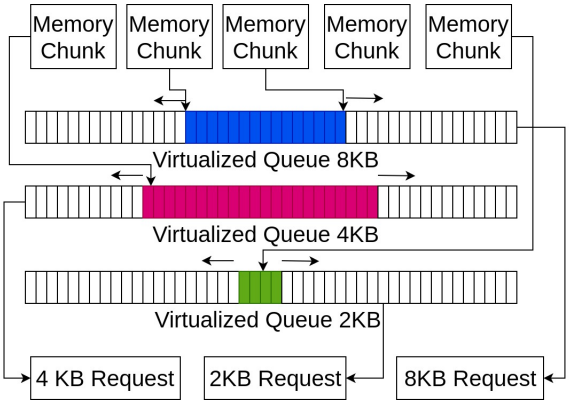


Figure 2: Ouroboros's design [36]: memory chunks are used to extend virtualized queues upon allocation requests. Multiple queues are maintained, each serving requests of different sizes

Figure 2 illustrates Ouroboros' design. Ouroboros divides the managed memory region into multiple queues, each serving a different page size. 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 the when it actually needs more space. In this design, the authors avoided pre-allocating memory for all the queues. By using a semaphore, Ouroboros suffers from the long latency in accessing queue states by concurrent threads. 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.

3 PARALLEL MEMORY ALLOCATION

3.1 Core Idea

We elaborate on our idea by first assuming the GPU global memory is divided into pages of equal size, and each memory allocation requests exactly one such page (we will relax this assumption in Section 4.3). The core technique is a Random Walk (RW) algorithm that does not depend on any global

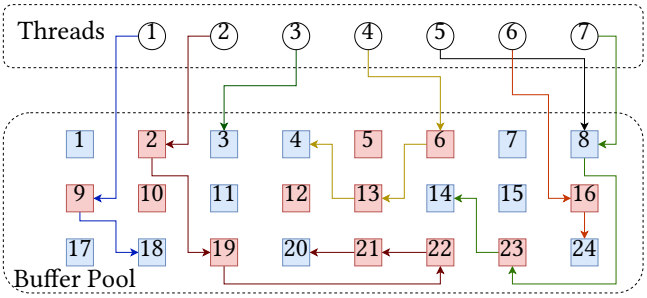


Figure 3: A case of the RW-based page request algorithm. The path visited by the same thread is colored the same. Blue pages are free, red pages are occupied.

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 ID until 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 3 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 repetitively generates random page IDs until it finds a free page. If more than one threads select a page at the same time, one of them can acquire the page through an atomic operation while the others continue their search without waiting. The chance of this happening is low as proven in section 3.2.2.

Detailed implementation of the RW-based memory allocation (we name it GETPAGE) 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.

By the first look, RW is counter-intuitive: the number of steps to get a free page can be big (e.g., 5 steps for thread 2). However, we show that the the average number of steps taken by all threads is highly controllable under most scenarios. Our analysis in Section 3.2 will clearly show this.

Deallocation: A great advantage of our method is: the deallocation (named FREEPAGE) is almost free! Specifically, we only need to clear the *used* bit of the corresponding page.

3.2 Performance Analysis

Since there will be many notations in this section, common notations are summarized in Table 1 for easier reference.

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
```

Table 1: Common notations

Notation	Meaning
T	Total number of pages
A	Total number of free pages
N	Number of threads
X_i	Number of steps for thread i to find a free page
Y	Maximum number of steps to find a free page within a warp
TAS, $E(X_i)$	Per-Thread Average Steps
WAS, Y	Per-Warp Average Steps
w	Number of bits on the bitmap that can be read by a single instruction

3.2.1 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 actual running time of a CUDA function is affected by many factors. 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. While the latency of one step is not the same for all designs and implementations, we choose to focus on the number of steps in this analysis because it is a major indicator of scalability.
2. **Per-Warp Average Steps (WAS):** the average of the maximum number of steps taken among all threads in a warp.

Both metrics are directly correlated to latency. 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. Thus, WAS is a better indicator of total running time, yet we achieve more rigorous analysis of TAS.

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.

Due to page limits, we will focus on the analytical results while skipping the proof.

First of all, we can easily show that in a queue-based solution such as Ouroboros, the TAS is $E(X_i) = \frac{N+1}{2}$ and $E(Y)$ can be derived by the Faulhaber's [22] formula as:

$$E(Y) = N - \frac{\frac{N^{33}}{33} + \frac{N^{32}}{2} + \frac{8N^{31}}{3} - \frac{124N^{29}}{3} + \dots}{N^{32}} \approx \frac{32}{33}N \quad (1)$$

Both metrics are linear to N, consistent with results 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.

3.2.2 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 3, 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.

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 [8] to approximate the harmonic series $H_n \approx \gamma + \ln n$. The expected average number of steps is then approximated by

$$E(X_i) \approx \frac{T}{N} ln\left(\frac{A}{A-N}\right) \quad (2)$$

Unlike the queue-based solution with latency linear to N, Eq. (2) 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 large TAS is when $A \approx N$, i.e., when there are barely enough pages available for all the threads.

The above analysis can be verified in Figure 4(a) where we plot the value of formula (4) 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.

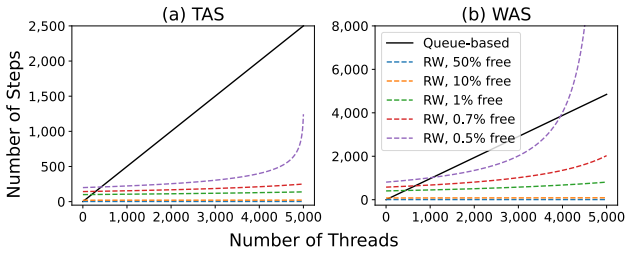


Figure 4: Change of TAS (a) and WAS (b) values under different N and A values of the RW-based algorithm in comparison to that of a queue-based solution

Independence Among Threads: The above analysis did not consider the scenario in which multiple threads try to atomically acquire a free page at the same time. As a result, actual TAS values can be larger due to such collisions. This can be modeled via 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 [26]). This number should be small in any reasonable setup, e.g., 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). Hence, the expected number of collisions is further bounded by

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

3.2.3 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 $p = \frac{A-N}{T}$.

Since $E(X_i)$ is upper bounded by $E(X_i')$, $E(Y)$ is also upper bounded by $E(Y')$ where $Y' = max(X_i')$ as follows:

$$E(Y) < \sum_{k=0}^{\infty}\left[1 - \left(1 - \left(\frac{T-A+N}{T}\right)^k\right)^{32}\right] \quad (3)$$

In Figure 4(b), we plot the calculated values of the RHS of Eq. (3) with $T = 1M$. Obviously, this bound is larger than $E(X_i)$ (Figure 4(a)) under the same parameters. Same as $E(X_i)$, the bound of $E(Y)$ is still significantly smaller than the queue-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 4.2.

4 EXTENSIONS

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. In this section, we present two advanced techniques that address the above two issues (Sections 4.1 and 4.2). Furthermore, such design allows efficient implementation of functions that request memory of an arbitrary size (Section 4.3).

4.1 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 (details skipped).

4.1.1 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 $1 - (\frac{T-A}{T})^w$. Following the same logic in deriving Eq. (2) and Eq. (3), we get:

$$E(X_i) = \frac{1}{N}\sum_{j=0}^{N-1}\frac{1}{1-\left(\frac{T-A+j}{T}\right)^w} \tag{4}$$

In fact, Eq. (3) is a special case of Eq. (2) with $w = 1$, and the following theorem shows their difference.

Theorem 1. *Denote the TAS $E(x)$ for RW-BM as U', and that for the basic RW algorithm as U, we have*

$$\lim_{A\to N} U' = \frac{U}{w}$$

Proof. The proof is achieved via the Euler-Maclaurin formula [1], details skipped. □

Similarly, the upper bound of WAS in RW-BM becomes

$$E(Y) < \sum_{k=0}^{\infty}\left[1-\left(1-\left(\frac{T-A+N}{T}\right)^{wk}\right)^{32}\right] \tag{5}$$

and we also get the following theorem.

Theorem 2. *Denote the upper bound of $E(Y)$ for RW-BM as V', and that for the basic RW algorithm as V, we have*

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

The above theorems are encouraging: both TAS and the WAS bound decrease by a factor up to w, i.e., 32/64. Plus, the advantage of RW-BM is the highest when $A \to N$, which is the 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 4.2).

Memory Overhead: 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.

4.2 Collaborative Random Walk Algorithm

The basic RW design suffers from the large difference between TAS and WAS. To remedy that, our idea 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 in a warp. The algorithm repeats two steps: (1) the threads work together to find free pages and (2) the identified free pages are assigned to individual threads according to their needs. All threads participate in both steps, this leads to fewer iterations and the values of TAS and WAS become the same.

Efficient implementation of the above idea is non-trivial. The main challenge is to keep track of the found pages and distribute them to requesting threads in a parallel way. We design a Collaborative Random Walk (CoRW) algorithm (Algorithm 2) by taking advantage of CUDA *shuffle instructions* that allow access of data stored in registers by all threads in a warp, and other *intrinsic* functions. By working on data sit in registers, all such functions have very low latency thus ensure a good tradeoff between more instructions and less memory access (i.e., reduced WAS).

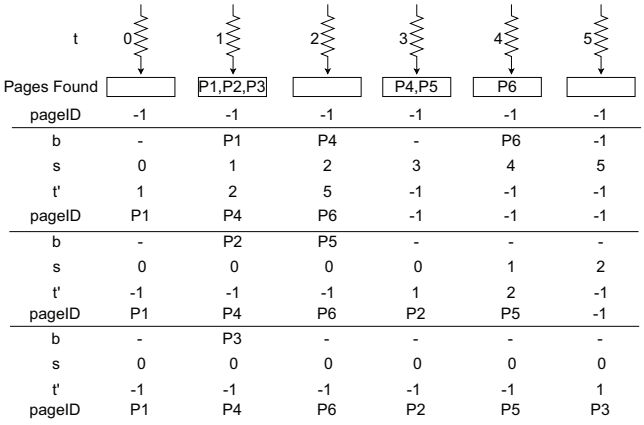
In Algorithm 2, *needMask* is a 32-bit mask representing what threads still need to get a page and *hasMask* those that find a free page during the search process. We perform the search process on all threads until all threads have obtained a page, i.e., *needMask* becomes 0 (line 4). The repeated search process is as follows. First, each thread reads a random word of the bitmap (line 7) denoted as BitMap[p]. 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 6). 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 share the free pages among threads (lines 10 to 16) while some thread still need a page and some thread still has a page to share (line 9). This is difficult because the CUDA shuffle instructions only allow a thread to read data from another thread, i.e., the receiving threads have to initiate the transfer. Therefore, our solution is to calculate the sending lane ID *t'* as follows. Each thread calculates *s*, the number of threads with lower lane ID that still need to get a page as indicated by *needMask* (line 12). Then this thread has to obtain the (*s+1*)-th page found within the warp because the first *s* pages should be given to the lower lanes. Therefore, *t'* is the position of the (*s+1*)-th set bit on *hasMask*. Here fks is a function for finding the k-th set bit by using $\log(w)$ population count (*popc*) operations (details skipped). This allows us to calculate the sending thread *t'* (line 13). Finally,

Algorithm 2: Collaborative getPage within a warp

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

1: t ← lane ID (0-31) of this thread
2: pageID ← -1
3: needMask ← *ballot_sync*(pageID==-1)
4: **while** needMask **do**
5: p ← random integer within $[0, T/w)$
6: r ← Atomically set LockMap[p] to 1
7: P ← (r == 1)? 0xffffffff : BitMap[p]
8: hasMask ← *ballot_sync*(P≠0xffffffff)
9: **while** needMask≠0 and hasMask≠0 **do**
10: Find the first 0 bit on P and set it to 1
11: b ← corresponding pageID of the bit set above
12: s ← *popc*(~(0xffffffff<<laneID) & needMask)
13: t' ← pageID==-1? *fks*(hasMask, s+1)-1 : -1
14: pageID ← $shfl\_sync(b, t')$
15: needMask ← *ballot_sync*(pageID==-1)
16: hasMask ← *ballot_sync*(P≠0xffffffff)
17: **end while**
18: Release LockMap[p] if r==0
19: **end while**
20: return pageID



pageID	-1	-1	-1	-1	-1	-1
b	-	P1	P4	-	P6	-1
s	0	1	2	3	4	5
t'	1	2	5	-1	-1	-1
pageID	P1	P4	P6	-1	-1	-1
b	-	P2	P5	-	-	-
s	0	0	0	0	1	2
t'	-1	-1	-1	1	2	-1
pageID	P1	P4	P6	P2	P5	-1
b	-	P3	-	-	-	-
s	0	0	0	0	0	0
t'	-1	-1	-1	-1	-1	1
pageID	P1	P4	P6	P2	P5	P3

Figure 5: Step-by-step (from top to bottom) changes of key variables in 6 threads running CoRW.

the value of variable b hold by thread t' is transferred to this thread via the *shfl_sync* function (line 14).

Figure 5 shows an illustrative example with 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 in a word (*popc*) requires only 2 clock cycles and execution of *fks* can be done in the low tens of cycles. Such latency is in sharp contrast to reading the bitmap from the global memory, which requires a few hundred cycles [3, 16].

Algorithm 3: RW_malloc

input : n, number of consecutive pages to find
output: the ID of the first free page

1: t ← lane ID (0-31) of this thread
2: pageID ← -1
3: needMask ← *ballot_sync*(pageID==-1)
4: **while** needMask ≠ 0 **do**
5: If t == 0 then p ← random integer within $[0, T/w)$
6: P ← BitMap[p+t]
7: S, L ← first range of consecutive unset bits on P
8: hasMask ← *ballot_sync*(L>0)
9: **while** hasMask≠0 and needMask≠0 **do**
10: **for** t' : 0 → 31 **do**
11: S1 ← $shfl\_sync(S, t')$
12: L1 ← $shfl\_sync(L, t')$
13: If pageID==-1 and L1>=n then take←true
14: **if** this has the lowest t whose take==true **then**
15: Atomically set the s1→S1+L1-1 bits of BitMap[p+t']
16: If success then pageID ← corresponding page ID
17: **end if**
18: **end for**
19: needMask ← *ballot_sync*(pageID==-1)
20: S, L ← next consecutive unset bits on P
21: hasMask ← *ballot_sync*(L>0)
22: **end while**
23: **end while**
24: return pageID

4.3 Allocating Multiple Consecutive Pages

Our work on RW-BM and CoRW paved the way towards allocation consecutive memory of an arbitrary size (we name the function as **RW_malloc**). We still divide the memory pool into small units of the same size and store the *used* bits in a bitmap. 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.

RW_malloc can be viewed as an extension of CoRW (detailed design in Algorithm 3 and an illustrative example in Figure 6). However, instead of each thread's reading a random word, all (active) threads in a warp will read in consecutive words to get a large region (e.g., 4096 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). Free segments running across two neighboring words are also connected. Critical information (e.g., starting position, length, used or not) of all free segments are stored in local registers. Finally, threads use shuffle instructions to read all

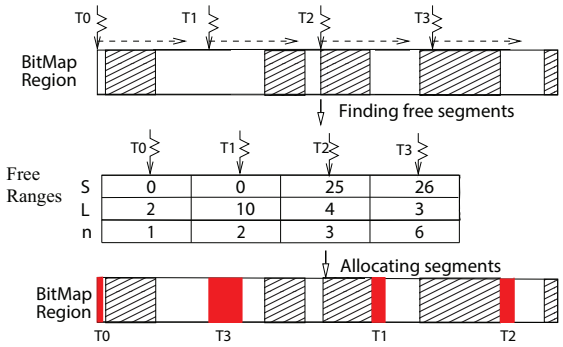


Figure 6: An example of 4 threads running RW_malloc.

free segments found within the warp and find a free segment that can serve its RW_malloc request.

As compared to CoRW, the main innovation is finding a range (S, L) of consecutive unset bits on a word (line 7 and 22) where S is the starting position and L is the length of the unset bits. Here we only need to find the first of such ranges, share it among the warp, find the next range, and repeat. Finding the first unset range can be done very efficiently by using the intrinsic function Count Leading Zero (*clz*) as follows. First, we complement the word and use *clz* to find the number of leading *set* bits. Next, we unset the leading set bits and use *clz* again to find the number of leading *unset* bits. Finally, S is equal to the number of leading set bits, and L is equal to the difference of the leading unset bits and the leading set bits. Sharing ranges among a warp is performed in a similar manner as in CoRW.

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 because we have to read all segments in a warp to find a suitable one. However, since all data is stored in registers and shared by shuffle instructions, the overall performance is still much higher than Ouroboros.

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.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We perform experiments to compare our methods with Ouroboros. We use the same Ouroboros code and environment configurations as presented in [36] to ensure a fair and meaningful comparison. In all experiments, we configure both Ouroboros and our systems to have a total of 8 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.

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 20.04 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.

5.2 Experimental Results

5.2.1 Performance of GetPage. First, we evaluate the performance of the two methods in getting a single fixed-size page. Specifically, we develop a single GPU kernel whose only task is to request a page. The equivalent in the Ouroboros system is a kernel that requests 256B for each thread because 256B is our page size. We launch the kernel with various numbers of threads and various percentages of free pages.

Figure 7 shows the three metrics (kernel running time, TAS, and WAS) measured from kernels as well as theoretical values of TAS/WAS generated from our analyses. 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 CoRW outperforms Ouroboros by more than an order of magnitude under most of the cases. When there are less than 1% free pages, the advantage of our method starts decreasing, but the CoRW still outperforms Ouroboros by a big margin. Note that the 1% free page scenario 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. Ouroboros' running time increases with number of threads but our algorithm is insensitive to that (except under 0.5% free page).

Results from the second and the third rows confirm the validity of our theoretical results (i.e., Equations (2) to (5)). First, the measured TAS values match the theoretical results well.

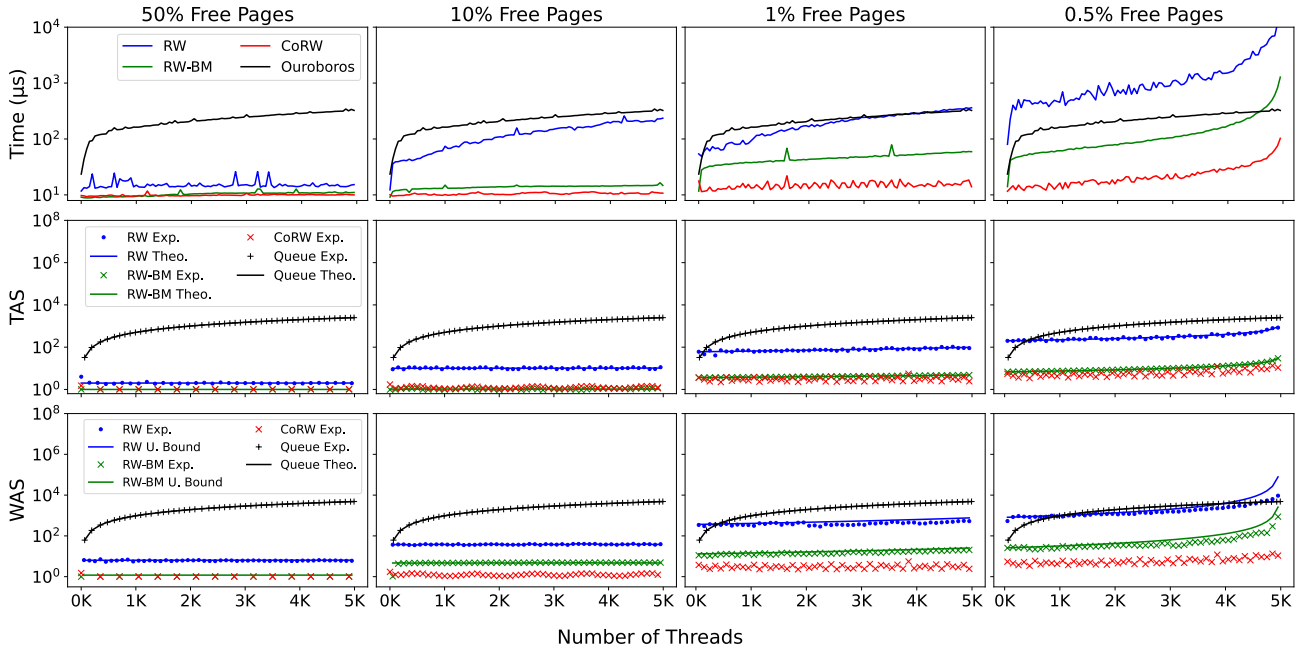


Figure 7: Performance of the kernel calling getPage under different numbers of parallel threads and percentage of free pages. Percentages of free pages are measured at the start of each kernel.

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% .

5.2.2 Performance of RW_malloc. Here we perform three experiments matching those in [38]. First, we compare RW_Malloc and Ouroboros as the number of threads increases. Second, we compare them as the allocation size increases. Third, we compare them with allocation requests of mixed sizes. All experiments have the same setup as in [38] except that allocated pages are not immediately freed after the allocation. This change affects Ouroboros significantly because it has to keep extending its queues. However, this change reflects real-world applications better than the original setup.

Figure 8 presents the total kernel time of RW_malloc and Ouroboros when the number of threads scales. In allocation, RW_malloc is faster than Ouroboros by up to 3 orders of magnitude. The improvement is higher as the number of thread grows, except for the 8192-byte allocation because 2^{20} allocations of 8192 bytes deplete the memory pool and bring our system to its worst scenario. Similarly, RW_malloc performs better in freeing memory as the number of threads grows and by up to an order of magnitude.

Figure 9 shows the total kernel running time when the allocation size scales. Ouroboros' performance decreases by more than an order of magnitude when allocation size increases from 4 to 8K bytes. RW_malloc is insensitive to the change of allocation size.

Figure 10 presents the total kernel time in allocating mixed sizes. This figure shows that RW_malloc outperforms Ouroboros in allocating a wide range of memory sizes. The improvement reaches 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. The only case where Ouroboros wins is when it immediately frees memory just allocated. By this, Ouroboros hits a sweet spot since it does not need to allocate new chunks to extend the virtualized queues. However, we believe this is an unrealistic scenario, as buffers will normally be used before released.

For RW_malloc, we also evaluated memory utilization. Same as in [38], we keep sending allocation requests until a system reports an out-of-memory error. The memory utilization is calculated as the fraction of total allocated amount in the entire memory pool. 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. According to Figure 11, memory utilization of RW_malloc is always close to 100%. On contrary, Ouroboros performs well when allocation sizes are

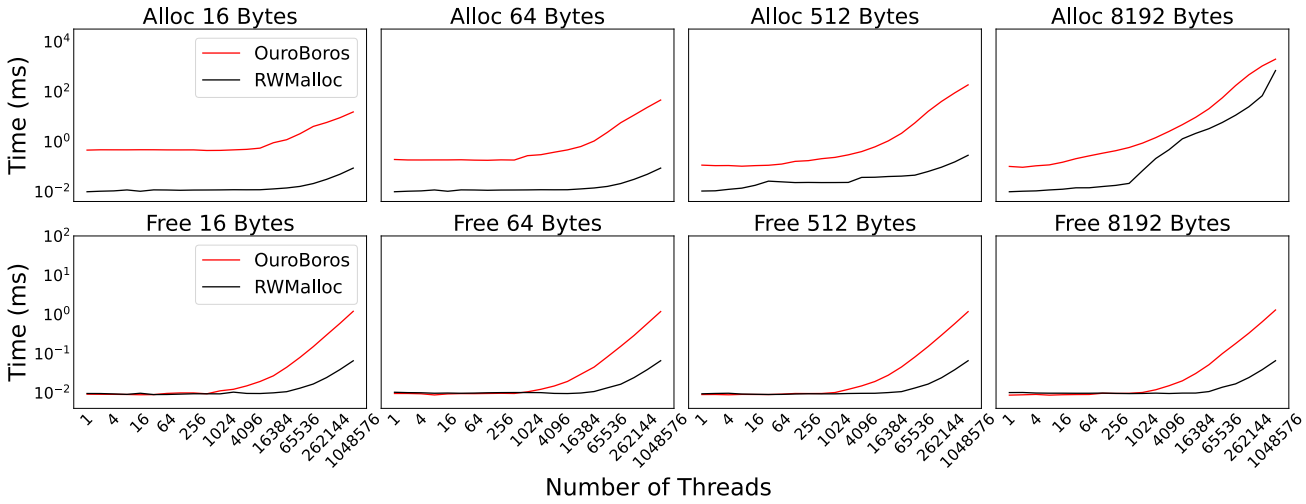


Figure 8: Performance of RW_malloc and Ouroboros under different numbers of threads.

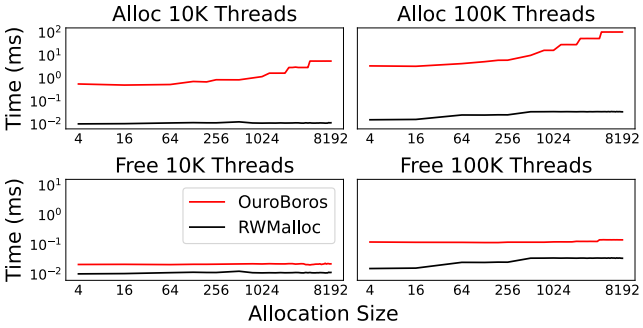


Figure 9: Performance of RW_malloc and Ouroboros under different allocation sizes.

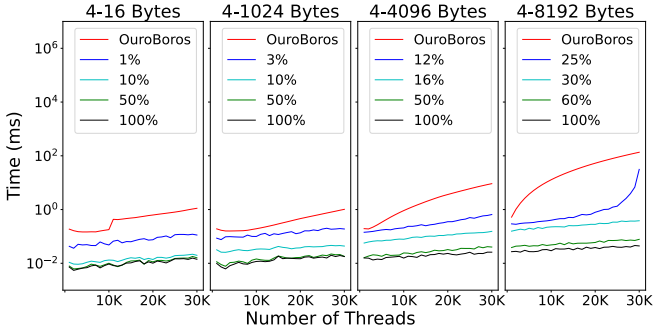


Figure 10: Performance of RW_malloc and Ouroboros in allocating mixed sizes.

some power of two because these sizes fit perfectly into the pages. The reason is that our system allocates large memory chunks by aggregating small consecutive pages and thus has a finer-granularity control over the memory space.

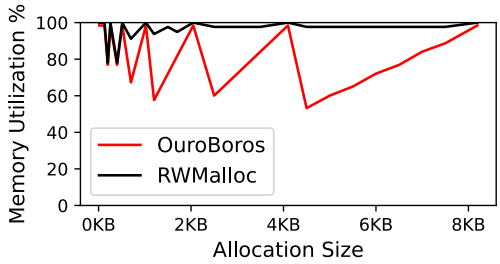


Figure 11: Memory efficiency of Ouroboros and RW_Malloc

5.2.3 A Case Study: Hash Join on GPUs. We report experimental results of real GPU programs with page acquisition needs served by our GETPAGE code. As compared to unit-tests, this gives us a chance to evaluate our methods in real-world applications. In particular, we focus the *probing* kernel of a state-of-art GPU hash join code [27] as the foundation of this case study. The kernel compares tuples of the corresponding partitions of the two tables and outputs one tuple whenever there is a match. We measure 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. As a baseline, we also include the original code used in [27] 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.

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 12, by using Ouroboros, the join kernel runs slower than others by a large margin. Ouroboros has the worst

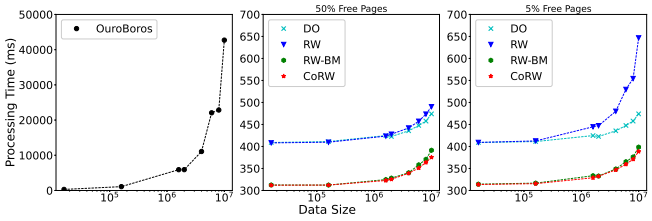


Figure 12: Total running time of a GPU hash join program enhanced with different buffer management mechanisms under different input table sizes

performance and DO is slightly better than RW since it only involves a single *atomicAdd* operation. With the help of bitmap implementation, RW-BM and CoRW alogorithms outperformed DO in all cases.

6 RELATED WORK

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.* [21] brought up the idea of paging in Atlas system. Page sgementation was firstly discussed by Dennisó *et al.* [13]. Then Corbató *et al.* [12] 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 [30] that discussed OS supports in the context of a DBMS. Effelsberg *et al.* [14] discussed database buffer manager as a component of DBMS and implemented it. Chou *et al.* [11] presented a DBMIN algorithm to manage the buffer pool of an RDBMS. Chen *et al.* [10] proposed a query execution feedback model to improve DBMS buffer management. Brown *et al.* [7] introduced the concept *hit rate concavity* and developed a goal-oriented buffer allocation algorithm called Class Fencing.

Memory Management on GPUs: NVIDIA initially announced its dynamic memory allocator for GPUs in 2010 [38]. It provides the usual malloc/free interface and can be called by threads from a CUDA kernel. XMalloc [20] 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 [29] addressed this bottleneck by scattering the allocation requests across its memory regions. A hash function is used to search for free regions. FDGMalloc [34] (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 [2] 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) [32] proposed RegEff, which splits the bookkeeping 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.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we study the problem of runtime memory allocation in highly parallel software systems, with a focus on GPUs. The main idea of our design is to avoid maintaining metadata that could become a major bottleneck in a high-concurrency systems. Based on that philosophy, we propose a memory 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. To address the RW's limitations in extreme cases when free buffer pages are very rare, we propose two advanced techniques: the first one 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 involves the sharing of free pages among neighboring threads. We also present our solution to the problem of allocating arbitrary bytes. Our experimental results are consistent with the mathematical analyses. The results show that our solutions significantly outperform the best known GPU memory allocator, Ouroboros, in both allocation/deallocation performance and memory utilization.

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.

REFERENCES

[1] Milton Abramowitz and Irene A Stegun. [n.d.]. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. New York: Dover Publications, 16,806,886.

[2] Andrew V Adinetz and Dirk Pleiter. 2014. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. In *GPU Technology Conference (GTC)*, Vol. 152.

[3] Yehia Arafa, Abdel-Hameed Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Low Overhead Instruction Latency Characterization for NVIDIA GPGPUs. arXiv:1905.08778 [cs.DC]

[4] Iya Arefyeva, David Broneske, Gabriel Campero, Marcus Pinnecke, and Gunter Saake. 2018. Memory management strategies in CPU/GPU database systems: A survey. In *International Conference: Beyond Databases, Architectures and Structures*. Springer, 128–142.

[5] Toufik Baroudi, Vincent Loechner, and Rachid Seghir. 2020. Static versus Dynamic Memory Allocation: a Comparison for Linear Algebra Kernels. In *IMPACT 2020, in conjunction with HiPEAC 2020.*

[6] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices* 35, 11 (2000), 117–128.

[7] Kurt P Brown, Michael J Carey, and Miron Livny. 1996. Goal-oriented buffer management revisited. *ACM SIGMOD Record* 25, 2 (1996), 353–364.

[8] Tomislav Burić and Neven Elezović. 2013. Approximants of the Euler–Mascheroni constant and harmonic numbers. *Appl. Math. Comput.* 222 (2013), 604–611.

[9] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[10] ChungMin Melvin Chen and Nick Roussopoulos. 1998. *Adaptive database buffer allocation using query feedback*. Technical Report.

[11] Hong-Tai Chou and David J DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica* 1, 1-4 (1986), 311–336.

[12] Fernando J Corbató and Victor A Vyssotsky. 1965. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*. 185–196.

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

[14] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 560–595.

[15] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking energy. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 393–406.

[16] Minquan Fang, Jianbin Fang, Weimin Zhang, Haifang Zhou, Jianxing Liao, and Yuangang Wang. 2018. Benchmarking the GPU memory at the warp level. *Parallel Comput.* 71 (2018), 23–41.

[17] Wolfram Gloger. 2006. Wolfram Gloger's malloc homepage.

[18] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–39.

[19] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[20] Xiaohuang Huang, Christopher I Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. 2010. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 1134–1139.

[21] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. 1962. One-level storage system. *IRE Transactions on Electronic Computers* 2 (1962), 223–235.

[22] Donald E Knuth. 1993. Johann Faulhaber and sums of powers. *Math. Comp.* 61, 203 (1993), 277–294.

[23] Tongping Liu, Charlie Curtsinger, and Emery D Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 327–336.

[24] NVidia. 2018. CUDA C Programming Guide. Retrieved Febrarury 24, 2018 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[25] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1935–1950.

[26] Igor Pletnev, Andrey Erin, Alan McNaught, Kirill Blinov, Dmitrii Tchekhovskoi, and Steve Heller. 2012. InChIKey collision resistance: an experimental testing. *Journal of cheminformatics* 4, 1 (2012), 1–9.

[27] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (Chicago, IL, USA) *(SSDBM '17)*. ACM, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[28] Matthias Springer and Hidehiko Masuhara. 2018. DynaSOAr: a parallel memory allocator for object-oriented programming on GPUs with efficient memory access. *arXiv preprint arXiv:1810.11765* (2018).

[29] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–10.

[30] Michael Stonebraker. 1981. Operating system support for database management. *Commun. ACM* 24, 7 (1981), 412–418.

[31] RD Team et al. [n.d.]. RAPIDS: Collection of Libraries for End to End GPU Data Science, 2018.

[32] Marek Vinkler and Vlastimil Havran. 2015. Register efficient dynamic memory allocator for GPUs. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 143–154.

[33] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.

[34] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. 2013. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th workshop on general purpose processor using graphics processing units*. 120–126.

[35] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*. Springer, 1–116.

[36] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. 2020. Ouroboros: virtualized queues for dynamic memory management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[37] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. 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*. IEEE, 754–766.

[38] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. 2021. 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*. 219–233.

[39] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 44.

[40] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.

[41] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. 2013. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1374–1377.