Host-Based Allocators for Device Memory

1st Oren Bell
Washington University in St Louis
St Louis, USA
oren.bell@wustl.edu

2nd Ashwin Kumar
Washington University in St Louis
St Louis, USA
ashwinkumar@wustl.edu

3rd Chris Gill
Washington University in St Louis
St Louis, USA
cdgill@wustl.edu

***Abstract*—Segregated Fit and Two-Level Segregated Fit are two common dynamic memory allocation algorithms used in real-time systems, due to their constant-time allocation and deallocation of memory.**

We pose a model where these allocation algorithms run on a host system but manage device memory. This model is exemplified in accelerated applications without unified memory that copy data in and out of device memory before and after a kernel call. Managing device memory from the host device incurs the following constraint: the allocator can't read the memory it is managing. This means we are unable to use boundary tags, which is a concept that has been ubiquitous in nearly every allocation algorithm, including segregated fit and two-level segregated fit. In this paper, we propose alternate designs to work around this constraint, and discuss in general the implications of this system model.

***Index Terms*—GPU, FPGA, hardware acceleration, heterogeneous computing, memory management**

I. Introduction

This paper concerns itself with real-time dynamic memory management, i.e., algorithms to manage a heap of memory that clients can request and free blocks from at any time, and do so in O(1) time. The field of dynamic memory management could be said to have been started by Knuth [1]. Some 30 years of progress is well summarized by Wilson and Johnstone [2][3] in their comprehensive survey. Since then, further improvements have been made [4][5][6][7], but overall the subject can be considered to be highly mature.

Most existing memory management algorithms include metadata as headers or footers within allocated blocks. In contrast, our system model assumes that the allocator is running on a host system, managing device memory. This may occur, for example, if part of a computation is offloaded to a GPU or FPGA device while the rest of the computation runs on a multicore processor. This in turn implies that the compute device cannot (conveniently and efficiently) access the memory it manages. Therefore, any data needed by the allocation algorithm cannot be stored in its allocated blocks.

We motivate this system model by considering the usecases surrounding how device memory is managed. In the current state of the art, memory allocation/deallocation is either done on the peripheral compute device in often proprietary device drivers. This causes memory allocators to be treated as a fixed black box.

However, different applications may have differing needs to manage their memory. The most optimal allocator may be high performant, real-time constrained, or have domain-specific characteristics catering to the application. Fixing the allocator choice in proprietary drivers and hardware denies developers the choice to optimize this aspect of their program through selective mapping of portions of the application and its supporting libraries to different computational devices. Examples of such applications include drone cinematography [8] and real-time hybrid simulation experiments in earthquake engineering [9].

For our work, we assume that existing hardware and drivers are used to allocate arbitrarily large blocks of memory, but finer-grained memory allocation is then done in userspace by the host machine. As was mentioned previously, these allocators are constrained by the inability to read the memory they are managing and cannot store metadata in allocated blocks. We present alternative algorithms that overcome this constraint.

In Section II, we propose alternative measures to overcome this constraint of being unable to read managed memory. In Section III, we present our updated alternative allocation algorithms. We compare the performance of one of our algorithms to the default CUDA memory allocation functions in Section IV. Finally, we conclude in Section V with thoughts on implementations and usecases for this work.

II. Alternatives to Free Lists

Our paper explores a model in which device memory is managed from the host. This prevents the allocator from reading the memory being managed, which creates a new challenge not present in traditional memory allocation schemes. We cannot use boundary tags, a standard approach that was first mentioned by Knuth [1].

The traditional usage of boundary tags is illustrated in Figure 1. A free list is formed by a linked list of blocks that are available for reuse. The header points to the next element in the free list, and the footer points to the header of the same block, which enables coalescence between two adjacent blocks in O(1) time. After a block is freed, one can subtract the footer size from the block's address to obtain the address of the header of the prior block. From there, one can check if that prior block is in the free list, and if so, the two will be coalesced. The next block in the free list can also be checked to see if it's adjacent in memory, possibly coalescing a total of three blocks.

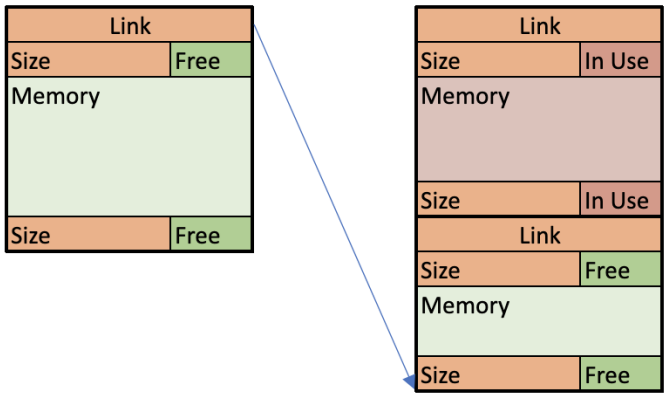

Fig. 1: Demonstration of Boundary Tags

A. Hash Tables

The strategy we propose is to use a hashtable, keyed by device memory address, to store all the information typically found in the header and footer of an allocated block.

The traditional algorithms for free-list traversal and coalescence operation would need to be modified to accommodate an extra step to lookup data in the hashtable. Additionally, the in-use blocks would also need to be tracked in the hashtable as well, as they also have necessary information in their header and footer.

With the use of hash tables, we aim to achieve O(1) allocation and deallocation, so co-mingling free and in-use blocks will not be a major performance concern.

The key for the hash table is the address of a block. The value contains i) the block size, ii) the address of the next block in a free list, iii) the address of a previous block in a free list, iv) the address of the previous block adjacent in address space. Thus we can create, in effect, free lists that span across a hash table. The ability to easily maintain multiple free lists is useful for segregated fit algorithms.

The primary downside of this approach is the overhead. Each entry contains 6 words (the key, block size, two references for a doubly linked list, one reference for a prior adjacent block, and a reference for a separate list linking collisions in the hash table). This means that in the worst case scenario, minimum allocation size of one word, the overhead is $\sim 83\%$. This is comparable to the worst case overhead of 80% seen in doubly linked free lists in conventional host-only algorithms. However, we only recommend this approach for applications with a fewer small allocations, such as those with a larger minimum block size, or any applications for which segregated fit algorithms are applicable.

B. Bitmasks

One way to minimize overhead is through the use of bitmasks. In this approach, device memory is divided equal-sized chunks, and the only overhead is a single bit in host memory to indicate whether that block is free or not. If we consider an allocation size lower bound of 8 bytes to be the worst case scenario, this creates an overhead of only 1.5% (one bit for 8 bytes). That produces 16MB of overhead for a gigabyte of device memory.

Allocation is done by finding a contiguous string of set bits in the bitmask corresponding to the desired size. For example, to allocate 1kB of memory, it is necessary to find a string of 128 bits that are all ones, incurring an O(n) cost for allocation, on the order of the address space size. The `clz` and `ffs` hardware instructions can be used to accelerate a bitsearch such as this.

All these bits must then be set to zero to indicate their corresponding blocks are in use. When the block is freed, they are all returned to ones (coalescence is implicit here). This means allocation and deallocation operations also incur an O(s) cost, on the order of the block size.

Bitmasks have low overhead and relatively high time complexity on the order of allocation size. Because of the low overhead, a potential use-case is to use bitmasks to manage a pool of small memory chunks, either of the same size (as in an object pool) or commingled varying sizes, as described above.

In order to achieve the real-time O(1) allocation and deallocation, the size of the bitmask must be bounded. For instance, a `clz` and `ffs` can search a 64-bit string in a single operation. This feature is also used in segregated fit algorithms that maintain multiple free-lists of different size classes. The bitmask serves as an availability mask to indicate which free-lists are non-empty.

III. Segregation Algorithms

A. Segregated Fit

In segregated fit, multiple free lists are maintained in size buckets for different powers of 2. Given a requested size to allocate, it is rounded up to the nearest order of magnitude, and then the first block from that list is selected. This means (assuming no empty free lists) the selected block may be nearly 4x larger than the requested size, causing 75% fragmentation, as discussed in prior literature [2] [3].

All lists are initialized pointing at a null block: an invalid block in the hashtable is meant to indicate the end of a free list. Accompanying this array of lists is an availability bitmap that can indicate which lists are non-empty. If the free list for a desired size class is empty, the next eligible list can be found by using the find-first-set (ffs) bit operation on the bitmap.

Our implementation stores the free lists in a hashtable, as discussed in Section II-A. Algorithm 1 and 2 show pseudocode for allocation and deallocation, respectively.

B. Two-Level Segregated Fit for Device Memory

Two-Level Segregated Fit[10][11][6] (TLSF) is a real-time dynamic memory allocation algorithm. Its primary benefits are reduced fragmentation and O(1) allocation and deallocation. It is an extension of segregated fit, dividing class sizes not only logarithmically, but also linearly, in a two-tiered system.

The allocation and deallocation steps are functionally the same, except the lookup step requires consulting 2 bitmasks.

Algorithm 1: Allocate memory in segregated fit

```
Data: free_lists, bitmask, hashtable, size ≥ 0
▷ Find available free list large enough to accommodate
requested_bins ← 1 ≪ ceil(log2(size));
order ← ffs(requested_bins);
                ▷ Lookup head of candidate list in hashtable
it = ht[free_lists[order]];
free_lists[order] ← it.next_free;
it.free = False;
                          ▷ Split off surplus portion of block
if size < it.size then
    new_block.size ← it.size − size;
    new_block.free ← True;
    new_block.addr ← it.addr + size;
    new_block.prev_adj ← it.addr;
    new_block.prev ← 0;
    it.size ← size;
       ▷ Place new block at head of free list in its size
     class bin_idx ← floor(log2(new_block.size));
    new_block.next = free_lists[bin_idx];
    new_block.prev = 0;
    ht[new_block.next].prev = new_block.addr;
    free_lists[bin_idx] = new_block.addr;
    bitmask ← bitmask | (1 ≪ bin_idx);
    ht[new_block.addr+new_block.size].prev_adj ←
     new_block.addr;
                                  ▷ Insert into the hashtable
    ht[new_block.addr] = new_block;
ht[it.addr] = it;
return it.addr;
```

Algorithm 2: Deallocate memory in segregated fit

```
Data: free_lists, bitmask, hashtable, addr ≥ 0
it ← ht[addr];
left ← ht[it.prev_adj];
right ← ht[it.addr + it.size];
if left.free then
    if right.free then
                          ▷ Coalesce all 3, erase it and right
        left.size+ = it.size + right.size;
        ht[right.addr + right.size].prev_adj ←
         left.addr;
                                          ▷ Update bitmask
        if right.prev = right.next then
            unset_bit ← 2^floor(log2(right.size));
            bitmask ← bitmask & ¬unset_bit;
        remove(right);
    else
                                ▷ Coalesce left block, erase it
        left.size+ = it.size;
        right.prev_adj ← left.addr;
    if it.prev_free = it.next_free then
        unset_bit ← 2^ceil(log2(it.size));
        bitmask ← bitmask & ¬unset_bit;
    remove(it);
    it = left;
else if right.free then
                                          ▷ Coalesce right block
    it.size+ = right.size;
    ht[right.addr + right.size].prev_adj ← it.addr;
                                          ▷ Update bitmask
    if right.prev = right.next then
        unset_bit ← 2^floor(log2(right.size));
        bitmask ← bitmask & ¬unset_bit;
    remove(right);
      ▷ Put coalesced block in free list in its size class
bin_idx ← floor(log2(it.size));
it.next = free_lists[bin_idx];
ht[it.next].prev = it.addr;
it.prev = 0;
free_lists[bin_idx] = it.addr;
it.free = True;
ht[it.addr] = it;
bitmask ← bitmask | 2^bin_idx;
```

One is logarithmic and functions just as detailed in Algorithm 1 and Algorithm 2. TLSF differs in that this points at another bitmask which subdivides the space linearly, as illustrated in Figure 2. This tier then points to a free list of blocks in that size class. If the free list is non-empty the linear bitmask will have a corresponding 1 set. If the linear bitmask is non-zero, then the logarithmic bitmask will have a corresponding 1 set.

Free list manipulations are done the same way as in segregated fit, so Algorithms 1 and 2 only need to be modified to account for this two-tiered lookup process and bitmask manipulation.

C. Hybrid Allocators and Object Buffers

Hybrid approaches may also be employed, using object buffers to manage object pools for allocations smaller than a page (<4kB), and segregated lists for larger allocations.

Object pools allow for the efficient management of vast amounts of small allocations, but are not a practical approach for larger allocations. A hybrid approach can permit the benefits of both approaches.

It is important to consider the practical use of such a hybrid approach in this context. A host-based allocator for device memory is unlikely to allocate large numbers of small objects, but rather large memory blocks that are passed to a kernel call for a hardware-accelerated device. If small allocations are created, they would be intermediary memory allocated by device code, likely using a SIMD allocator such as XMalloc[5] or ScatterAlloc[7]. These allocators are designed to handle many allocation requests in parallel without the need for locks, a usecase that doesn't apply to the host-based model we have presented in our paper.

So, our usecase would be best served by limiting the smallest granularity to a medium sized value, such as 4kB,

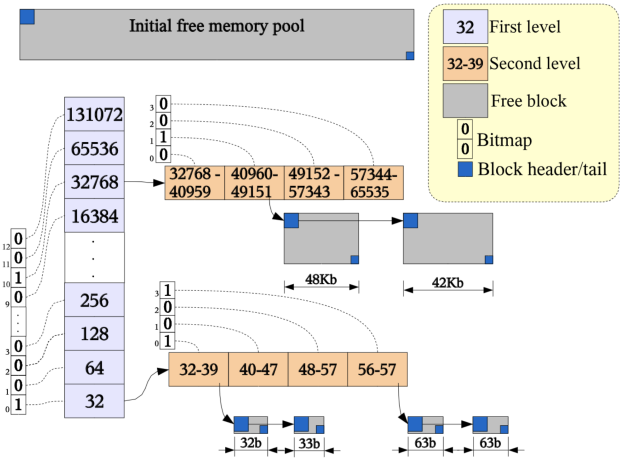

Fig. 2: TLSF Data Structure[11]

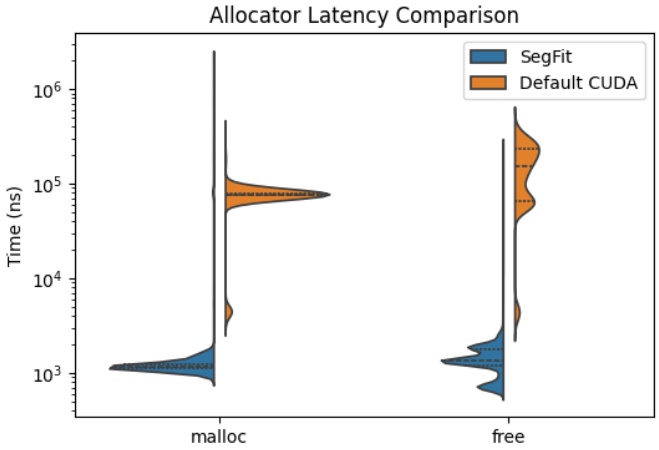

Fig. 3: Latency of malloc and free

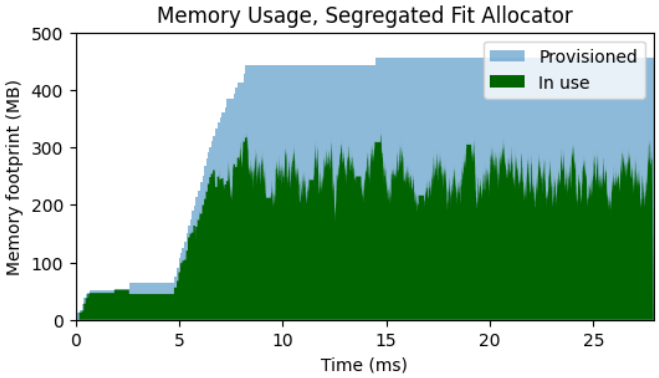

Fig. 4: Memory usage during benchmark

and exclusively using a segregated fit algorithm.

IV. Evaluation

To evaluate, we wrote an implementation of Algorithms 1 and 2. We make use of CUDA driver calls to perform initial allocation of physical memory when the memory pool doesn't have an adequate candidate block. After mapping the memory into virtual address space, it is treated as a contiguous and fungible memory pool. When a process terminates, all physical blocks are unmapped.

We compare our work to the default CUDA allocator. We adapted the malloc-large benchmark from the MiMalloc benchmark suite[12] to use an implementation of our segregated fit algorithm. This algorithm randomly allocates and frees blocks 1kB and 16MB in size, several thousand times.

As a baseline, we also ran the benchmark with ordinary calls to cudaMalloc and cudaFree. The order and size of the malloc and free operations was consistent between our work and the baseline. Latency comparisons between the two are shown in Figure 3.

Our work outperforms the default CUDA allocator by 2 orders of magnitude. There are some outliers during program initialization where the segregated fit malloc can take upwards of 2ms, but these disappear as the program settles into a steady state.

We measure fragmentation in our experiments (Figure 4) by tracking not only the memory in use by the benchmark, but also the physical device memory provisioned to the benchmark process. In our segregated fit allocator, we observe a roughly 50% fragmentation ratio. This is consistent with prior work[2] on segregated fit. A Two-Level Segregated Fit allocator would see improved fragmentation[6].

V. Conclusion

We note that prior work on dynamic memory allocation relies on free lists and boundary tags. We've posed a heterogeneous system model where memory is stored on one device and managed from another. After observing that existing algorithms are no longer applicable, we've provided alternatives to free lists and updated versions of classic sequential and segregated fit algorithms.

An implementation of our segregated fit algorithm was tested against conventional calls to cudaMalloc and cudaFree. It was found that, after a warmup period, our work consistently outperformed the CUDA calls by two orders of magnitude. This is mainly due to its implicit and efficient memory recycling.

We do not claim to outperform established GPU allocators [13] at scale, nor to address their highly parallelized use-cases. However, in specific circumstances where kernels are not expected to allocate memory and diverse accelerated computation is combined into a large heterogeneous application, performance benefits can be achieved by managing memory on the host. It is unnecessary to invoke device calls to free memory, and instead it can be dynamically recycled.

Acknowledgements

This work was sponsored by NSF Grant 2229290

REFERENCES

[1] D. E. Knuth, "The art of computer programming, vol 1: Fundamental," *Algorithms. Reading, MA: Addison-Wesley*, 1968.

[2] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[3] 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*. Springer, 1995, pp. 1–116.

[4] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger, "A compacting {Real-Time} memory management system," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[5] 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*. IEEE, 2010, pp. 1134–1139.

[6] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: A new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* IEEE, 2004, pp. 79–88.

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

[8] R. Bonatti, W. Wang, C. Ho, A. Ahuja, M. Gschwindt, E. Camci, E. Kayacan, S. Choudhury, and S. Scherer, "Autonomous aerial cinematography in unstructured environments with learned artistic decision-making," *Journal of Field Robotics*, vol. 37, no. 4, pp. 606–641, 2020.

[9] J. Condori, A. Maghareh, J. Orr, H.-W. Li, H. Montoya, S. Dyke, C. Gill, and A. Prakash, "Exploiting parallel computing to control uncertain nonlinear systems in real-time," *Experimental Techniques*, vol. 44, pp. 735–749, 2020.

[10] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, no. 2, pp. 149–179, 2008.

[11] M. Masmano, I. Ripoll, and A. Crespo, "Dynamic storage allocation for real-time embedded systems," *Proc. of Real-Time System Simposium WIP*, 2003.

[12] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 2019, pp. 244–265.

[13] 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*, 2021, pp. 219–233.