COALESCING MEMORY ALLOCATIONS FOR GPU MPI

Nils Blach, Martin Erhart, Florim Memedi, Anton Schäfer

Project Report
Design of Parallel and High-Performance Computing
ETH Zürich

ABSTRACT

We optimize in-kernel GPU memory allocations through static and dynamic coalescing of malloc requests on warp or block-level. This greatly benefits GPU MPI, a compiler that seamlessly ports distributed MPI programs to single GPU instances. MPI functions such as All-to-All, as well as distributed algorithms in general, perform many simultaneous memory allocations followed by repeated accesses, thereby exhibiting potential execution-time benefits from coalesced memory regions. Our allocator supported by static analysis and our universally applicable, dynamic allocator achieve over 4x speedup compared to NVIDIA's malloc implementation on an MPI All-to-All operation.

1. INTRODUCTION

Motivation. The Message Passing Interface (MPI) is a standard designed for massively parallel programs that is widely used in High Performance Computing (HPC). It provides a communication abstraction for running distributed programs on different infrastructure, usually large clusters. The GPU MPI compiler [1] translates MPI programs to CUDA code, allowing them to be run on single GPU instances that provide a simpler alternative to achieve massive parallelism.

As opposed to common CUDA programs, MPI based distributed algorithms often require dynamic memory allocation in the individual ranks, i.e., individual threads in GPU MPI. GPU MPI operations like All-to-All also rely on the allocation of global GPU memory in each individual thread. As a result, GPU MPI programs rely much more on dynamic in-kernel memory allocations than most CUDA programs.

An in-kernel dynamic memory allocator that coalesces such concurrent allocation requests can not only speedup allocation times by reducing synchronization overhead, but, more importantly, optimize memory layout, improving access efficiency. With memory access times being the bottleneck of many programs, this can yield significant speedups for GPU MPI workloads.

Related work. In recent years, many fast dynamic memory allocators for GPUs have been proposed. Most focus on an all-round efficient implementation, targeting some version of the design goals proposed by Steinberger et al. for ScatterAlloc [2]: correctness, speed (allocation time, deallocation time, access time, etc.) and memory consumption. FDGMalloc [3] uses the insight that threads in a warp rarely diverge in combination with so-called superblocks to achieve between one and two orders of magnitude speedup in allocation times over Scatteralloc. More recently, Galdo et al. [4] speed up dynamic allocation and de-allocation of resources through new synchronization primitives that match the capabilities of recent GPU architectures.

Contributions. We take a different approach, primarily optimizing work times by coalescing memory regions of adjacent threads, which can yield greater overall speedups. Our contributions are threefold: (i) We implement a static analysis that detects collective memory allocation in GPU MPI programs. (ii) We provide different allocators that coalesce the statically detected collective in-kernel memory requests. (iii) We implement a generally applicable allocator that dynamically detects and coalesces in-kernel memory requests and can be plugged into any CUDA program.

2. BACKGROUND

MPI is a standard proposed by a committee of vendors, faculty, implementors, and users. It defines communication environments and operations, specifically tailored for applications deployed on massively parallel infrastructures. With GPU MPI these infrastructures can be reduced to single GPU instances.

GPUs are processors based on the single-instruction-multiple-threads (SIMT) architecture optimized for executing the same instructions on large amounts of data in a highly parallel manner. They are widely used for graphics and machine learning workloads. A GPU consists of a grid of streaming multiprocessors (SM) that execute thread blocks which are subdivided into warps, i.e., groups of usually 32 threads that execute together.

When threads access memory, a memory transaction is issued and data is loaded from memory or cache. However, when threads in the same warp concurrently access

the same 32-byte segment, the memory transactions are coalesced into a single transaction [5]. Hence, by laying out data accessed by a warp compactly in memory, the number of memory transactions can be minimized. Additionally, the improved data locality can lead to more efficient caching.

The on-device dynamic allocator does not take full advantage of this.[1] No guarantees are given with respect to the offsets between memory regions returned to threads that are allocating collectively. We were unable to identify a consistent pattern by the allocator to achieve minimal offsets or coalesced memory regions. Furthermore, always at-least multiples of 80 bytes[2] are allocated, of which 16 bytes are reserved for an internally used header. This was verified by measuring the number of cycles needed to allocate 1 to 100 4-byte floats. We observed jumps every 20 floats, for instance allocating $1-16$ floats (4 to 64 bytes plus the 16 byte header) took $\sim 0.85 \times 10^6$ cycles each, whereas allocating $17-36$ floats took $\sim 2 \times 10^6$ cycles each. This not only limits the potential benefits of coalesced memory transactions for small allocation sizes, but also generally results in worse data locality.

3. OPTIMIZED ALLOCATORS

3.1. Static Coalescing

In this section we describe how static analysis can be utilized to perform coalesced memory allocations.

Static Analysis. To coalesce memory allocations across multiple threads, the allocator function needs to know how many bytes it is supposed to allocate. Checking this at runtime adds additional communication and computation overhead. Static Analysis can be utilized to detect memory allocations, performed by multiple threads, which can be coalesced.

In this project we distinguish the special case where all threads allocate the same amount of memory from the general case, where dynamic coalescing can be applied. A more elaborate static analysis would allow the static allocator to be used in more cases.

We implement our static analysis using the Clang AST matching and rewriting infrastructure. We use relatively simple patterns to match on malloc calls that are executed by all threads. These calls are replaced with calls to our own allocator which takes an optional argument that leads to coalesced memory allocation when set.

Functions that are part of the MPI program and have an accessible definition, i.e., not pre-compiled and linked against, are analyzed as well. When they perform some memory allocation, an optional function argument is added to indicate whether allocations inside of this function call may be coalesced. This argument can then be passed on to actual allocation calls.

Applying the match and rewrite patterns to the code colored black in listing 1 adds the parts in orange. The calls colored purple are not matched as they are not called by all threads at the same time. Additionally, at the beginning and end of the main function we add calls that allow our allocator to initialize and clean-up datastructures it might need, e.g., a hashmap.

All these additions are done entirely by the compiler without any user input. However, this does not hold for library functions not compiled with GPU MPI such as the MPI_Alltoall function. These have to be adapted manually to use our allocator and can then be registered in the rewriter to also use the optional coalescing argument.

As our allocator calls the standard CUDA-malloc only once for several threads in the coalesced case, each thread may not call standard CUDA-free individually. One option is to detect the corresponding malloc-free pairs in the MPI program and add the coalesce-argument. However, this is quite difficult to do in general as arbitrary pointer arithmetic can be performed in-between malloc and free calls. There can also be many free calls corresponding to one malloc nested in any combination of conditional and loop statements and we would have to guarantee to detect every single one of them. Thus we store whether it was a coalesced allocation in the allocator either explicitly or implicitly, e.g., using shared memory or alignment properties. This adds some runtime overhead to the free-calls, however, our main priority is to optimize memory access not de-allocation.

We do not support cases where a library function does some allocation and the MPI program is supposed to deallocate or vice versa. This is because incompatible malloc and free functions would be called.

```
void f(bool coalesce = false) {
  void *ptr = malloc(sizeof(int), coalesce);
  free(ptr);
}
int main(int argc, char **argv) {
  init_malloc();
  void *ptr = malloc(sizeof(int), true);
  f(true);
  int recv;
  MPI_Alltoall(ptr, 4, MPI_INT, &recv,
    4, MPI_INT, MPI_COMM_WORLD, true);
  free(ptr);
  if (threadIdx.x < 4) {
    void *ptr2 = malloc(sizeof(int));
    f();
    free(ptr2);
  }
  clean_malloc();
  return 0;
}
```

Listing 1. MPI program after static analysis and rewriting.

[1] Evaluated for compute capability 6.1

[2] Increases to some larger number if a thread allocates more than $320 - 16 = 304$ bytes.

Block-Level Coalescing. We design an allocator, which coalesces the allocated regions if the static analysis reports that all threads participate, so no runtime check has to be performed. In situations where static analysis fails to prove that all threads participate and allocate an equal number of bytes, our dynamic coalescing can be applied, as described in section 3.2.

The core idea of all the presented approaches is to have one leader thread allocate a continuous region, the superblock, using NVIDIA's malloc and distribute its sub-regions to all allocating threads.

To reduce overhead and complexity, we first focused on coalescing threads within the same block, as this allows the use of shared memory for communication and synchronization. To this end, we present two approaches, one that relies on headers to map sub-regions to respective superblocks and another that utilizes a separate, parallel hashmap instead.

Both allocators store a 4-byte counter at the beginning of each superblock, which is initialized by the leader to the number of sub-regions, respectively threads, contained. Whenever a thread frees its sub-region, it atomically decrements the counter. If the counter reaches value zero, the last thread to decrement will call NVIDIA's free using the superblock's base-address, as all sub-regions have been freed and it is save to deallocate the entire block. The two allocators differ in how the base-address of the superblock, and thus the counter, is determined.

The header based allocator stores a pointer to the superblock's counter directly in front of each sub-region returned to any thread, which can be used by the deallocator to perform the free. This has the downside of adding at least 8-bytes between each allocated region[3], which increases the offset between threads, potentially diminishing the benefits of coalesced memory transactions for small allocations and generally reducing data locality. This is circumvented by the hashmap based allocator, later referred to as no-header version, by maintaining the following mapping:

$$\text{sub-region address} \rightarrow \text{superblock base-address}.$$

The benefit of minimal offsets has the downside of increased malloc and free times, due to the extra hashmap insert and lookup/remove needed respectively, as well as a memory and initialisation overheads for the datastructure itself. Furthermore, distributing the sub-regions requires synchronizing all threads in the block adding an additional overhead to the malloc call for both allocators.

Warp-Level Coalescing. We also implement the hashmap based no-header allocator for warp-level granularity, so that allocations for all threads in a warp are coalesced within a superblock instead of all threads in a block. The differences are that more superblocks are used, thus more calls to NVIDIA's malloc and free are required, but less synchronization is needed.

[3]Note that potentially necessary padding was added to achieve the required 16-byte alignment

3.2. Dynamic Coalescing

Here, we motivate and outline our implementation of dynamic warp-level coalescing for the general case.

Idea. The described malloc implementations are applicable to the case where we can statically detect that all threads perform a malloc, requesting the same amount of bytes. For general programs, this is not always the case. Instead of falling back to the default malloc if the static coalescing methods are not applicable, we propose a malloc implementation that dynamically coalesces concurrent malloc requests and can be plugged into any CUDA program.

The dynamic coalescing is performed on the warp-level. There is little thread divergence within a warp; in some architectures, warps even execute in lock-step. This means, non-participating threads wait for completion of the others' allocation. Thus we expect to detect many threads participating in a malloc call while incurring low synchronization overhead for the non-participating threads. Warp-level synchronization is quite efficient in general, not only due to low divergence but also efficient builtin instructions.

Implementation. Our implementation uses superblocks containing small segments with headers and relies on warp-level synchronization and data shuffle instructions, completely avoiding the use of shared memory. A warp-level coalesced malloc consists of four main steps.

1. First, we detect which threads in the warp participate in the malloc call. Among these active threads, the thread with the lowest id is elected the leader.

2. Using warp-level shuffle instructions, all participating threads share their required memory sizes with the leader thread as well as all threads with higher ids.

3. The leader thread allocates the superblock and shares it with the participating threads via a warp-level shuffle.

4. By adding up the required payload and header sizes of the threads before, each thread then computes the offset of its memory segment in the superblock. The threads initialize their segment header and return a pointer to the payload.

To enable correct and wait-free free operations, each header contains its offset to the start of the payload of the previous segment as well as three special bits. The size of the previous segments enables finding the previous header without storing a full pointer. The three special bits are

`is_free` whether the segment has been freed already

`is_superblock` whether it is the first segment allocated by the leader thread

`is_last` whether it is the last / highest address segment that has not been freed yet

Using this information, free operates as follows: When a non-last segment is freed, we simply set the `is_free` bit to true and return. Only the last segment, i.e., the not-freed segment with the highest address among the segments allocated together, is responsible for freeing the superblock. Upon a free call to this segment, we either free the superblock or pass on the `is_last` flag (and the responsibility for freeing the superblock) to the new last segment. To this end, we walk along the chain of segments, linked by the size information in the headers. If all segments have been freed already, we eventually reach the first segment and free the entire superblock via standard CUDA-free. Otherwise, a non-free segment is reached and we set its `is_last` bit to true. Note that this has to be done with a compare and swap operation over the header bits to avoid race conditions when the segment is freed concurrently.

Optimizations. By default, we use eight byte headers, three bits of which are reserved for the special bits. However, especially for small allocation sizes, these headers incur a large space overhead and are not fully used. We avoid this by reducing the header size to four or two bytes if possible. If a header is e.g., four bytes long, we make sure the payload is not eight byte aligned but only four byte aligned such that the reduced header size can be detected by the free. This optimization allows for more compact data.

4. EXPERIMENTAL RESULTS

In this section we present the results of two types of benchmarks. Firstly, unit benchmarks that perform some simple workload such as summing up values on each thread individually and allocate the required memory directly in each thread. Secondly, we run MPI benchmarks which measure the runtime of MPI operations such as reduce, broadcast, and All-to-All using the GPU MPI compiler.

4.1. Unit benchmarks

Experimental setup. All unit benchmarks are run on an NVIDIA GeForce GTX 1050 Ti (compute capability 6.1, CUDA 11.3, driver 465.19.01) at 1.291 Ghz. We reserve 1 GB heap memory for memory allocation out of the 4 GB total memory, so there is enough room left to collect the timing measurements in the GPU memory.

We explore the performance gain for our different malloc implementations compared to the NVIDIA malloc implementation. Primarily, we are interested in the runtime when working with data on previously allocated memory. Additionally, we measure the duration of individual malloc and free calls on a per thread basis. For this we use in-kernel measurements around three separate parts: allocation, workload execution and freeing. Storing the cycle count using `clock64()` for each of these three parts per thread gives us precise measurements.

Each thread allocates memory for the same number n of single-precision floats. The allocated memory is then initialized with increasing values from 0 to $n-1$, allowing us to verify the expected results from later calculations, as opposed to using a random initialization. We present two out of five workloads with different computational complexities and memory access patterns in listing 2, i.e., the `Linear` and `Quadratic` workload.

```
Linear(n, data):
  sum = 0
  for i = 0..n-1:
    sum += data[i]
  data[0] = sum

Quadratic(n, data):
  sum = 0
  for i = 0..n-1:
    for j = 0..n-1:
      sum += data[i] * data[j]
  data[0] = sum
```

Listing 2. Workloads pseudo code. Results from the omitted workloads are similar.

The overall benchmark consists of multiple runs. In each run we measure allocation, workload execution and free time individually per thread. We then calculate the mean of these measurements over all threads, giving us three timings per run. Each run is repeated 20 times with two warmup runs. Further increasing the number of runs or warmups did not significantly influence our results.

```
_device_ Run(n):
  data = malloc_version(n floats)
  initialize(data)
  workload(n, data)
  free_version(data)
  // Return timings for malloc, work, free
  return timings

Benchmark(runs = 20, warmups = 2):
  for r = 1..runs + warmups:
    t = Run(n)<<<blocks, threads_per_block>>>

    // ignore warmup runs and aggregate
    m, w, f = mean_over_threads(t)
```

Listing 3. Unit benchmark pseudo code. The orange lines are individually timed.

Each run produces $3 \times total\_threads$ timing measurements. The total number of threads depends on the number of blocks and threads per block chosen for execution. We present results obtained with 12 blocks and 1024 threads per

block, allowing the hardware to be fully occupied. All other combinations with full hardware utilization yield consistent results. In the last line of listing 3 we calculate the means over all threads, giving us the three timings `m, w, f` per run for the respective timings of malloc, workload and free.

Results. Figure 1 and 2 show the performance on each workload of all five malloc variants for different numbers of floats to allocate per thread. Addition and multiplication are considered as one floating-point operation each. Both plots show measurements from 25 to 400 floats in steps of 25. The baseline shows the NVIDIA malloc implementation. On the linear workload we achieve the best performance with per warp allocation. We get speedups of more than 3x at 400 floats allocated. Figure 3 shows the relative speedups compared to the baseline.

On the quadratic workload the no-header variants yield the best performance. Note the absolute difference in performance compared to the linear workload. For both workloads we see a decrease in performance for larger allocation sizes. For allocation sizes below 25 floats we found that the per-warp variants perform similar to the baseline while the per-block variants perform worse.

Figure 4 shows timings for the allocation and free time. Our header + block implementation has consistently the best allocation and free times. Both per-warp variants have slower malloc and free times compared to the per-block variants. We hypothesize that this occurs due to the larger number of individual superblocks for the per-warp variants compared to the per-block variants. This may lead to more overhead on calling the NVIDIA malloc and free methods.

If we look at different combinations of blocks and threads per block, i.e., keeping the same number of total threads, we see larger allocation times when increasing the number of blocks. For a small number of floats we notice the overhead for creation of the hashtable, this overhead vanished before 100 floats.

4.2. MPI benchmarks

Experimental setup. All MPI benchmarks are run on an NVIDIA GeForce RTX 2060 (compute capability 7.5, CUDA 11.5, driver 495.44). We reserve 4 GB of heap memory of the 6 GB available. The setup for the MPI benchmarks is different from the unit benchmarks as GPU MPI does not support more than one thread per block for compute capabilities lower than 7.

In this benchmark we measured five workloads, namely the frequently used MPI functions Reduce, Broadcast, and All-to-All. The All-to-All benchmark was performed with 8 blocks with 32 threads each and 4 double-precision floating-point numbers to be exchanged. The number of blocks and threads is reduced because the heavy memory usage leads to infeasible runtimes for more threads. The other benchmarks

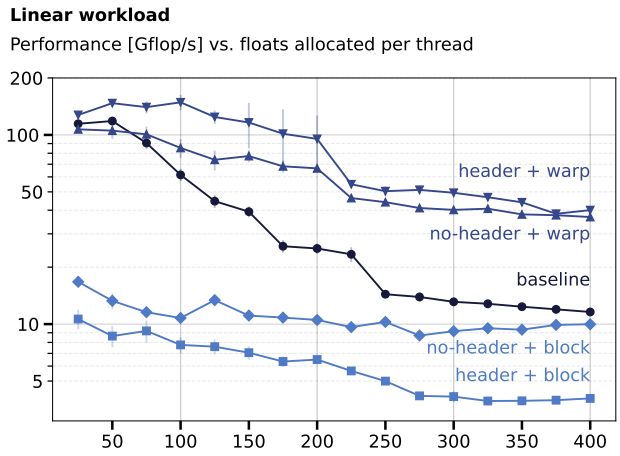

Fig. 1. Performance on the linear workload over 20 runs with 2 warmups.

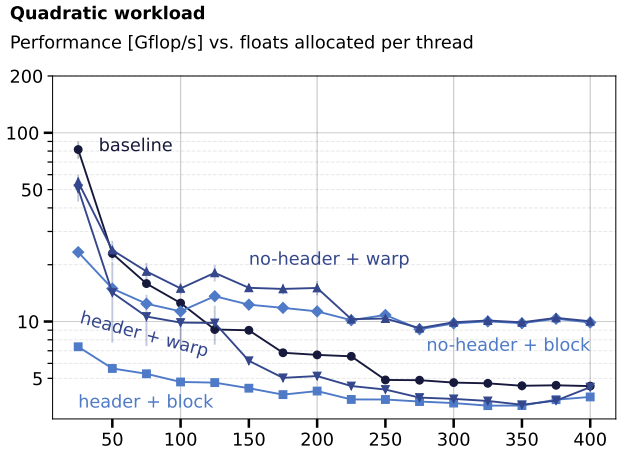

Fig. 2. Performance on the quadratic workload over 20 runs with 2 warmups.

Speedup compared to baseline (mean ± std)

		100	200	300	400
HB	L	0.13 ± 0.02	0.26 ± 0.02	0.32 ± 0.03	0.35 ± 0.02
	Q	0.38 ± 0.01	0.64 ± 0.03	0.78 ± 0.03	0.88 ± 0.03
NB	L	0.18 ± 0.02	0.42 ± 0.02	0.70 ± 0.03	0.86 ± 0.04
	Q	0.91 ± 0.03	1.70 ± 0.07	2.06 ± 0.07	2.18 ± 0.08
HW	L	$\mathbf{2.44 \pm 0.34}$	$\mathbf{3.81 \pm 1.28}$	$\mathbf{3.77 \pm 0.13}$	$\mathbf{3.46 \pm 0.23}$
	Q	0.79 ± 0.21	0.77 ± 0.03	0.82 ± 0.03	0.99 ± 0.03
NW	L	1.40 ± 0.21	2.66 ± 0.19	3.07 ± 0.19	3.18 ± 0.22
	Q	$\mathbf{1.20 \pm 0.10}$	$\mathbf{2.27 \pm 0.22}$	$\mathbf{2.09 \pm 0.08}$	$\mathbf{2.21 \pm 0.08}$

H: header, N: no-header, B: block, W: warp, L: linear, Q: quadratic

Fig. 3. Speedup for different number of floats allocated. Best speedup per column is bold.

Malloc time [ms]

	100	200	300	400
base	10.37 ± 0.36	57.30 ± 1.28	294.74 ± 5.46	296.95 ± 5.11
HB	$\mathbf{0.13 \pm 0.01}$	$\mathbf{0.20 \pm 0.03}$	$\mathbf{0.32 \pm 0.08}$	$\mathbf{0.35 \pm 0.10}$
NB	1.62 ± 0.01	1.63 ± 0.01	1.63 ± 0.01	1.61 ± 0.01
HW	1.09 ± 0.81	2.11 ± 1.73	7.28 ± 3.31	7.49 ± 3.29
NW	2.55 ± 0.44	4.49 ± 1.54	10.34 ± 4.51	9.01 ± 3.80

Free time [ms]

	100	200	300	400
base	0.06 ± 0.03	0.10 ± 0.04	0.57 ± 0.30	0.79 ± 0.42
HB	$\mathbf{0.01 \pm 0.00}$	$\mathbf{0.01 \pm 0.00}$	$\mathbf{0.02 \pm 0.00}$	$\mathbf{0.02 \pm 0.00}$
NB	0.07 ± 0.02	0.08 ± 0.02	0.08 ± 0.02	0.07 ± 0.02
HW	0.08 ± 0.02	0.15 ± 0.07	0.47 ± 0.20	0.46 ± 0.16
NW	0.86 ± 0.33	1.63 ± 0.62	1.27 ± 0.46	1.56 ± 0.71

H: header, N: no-header, B: block, W: warp

Fig. 4. Malloc and free time for different number of floats allocated. Values are mean $\pm$ std. Best time per column is bold.

used 30 blocks with 512 threads each to fully occupy the GPU and were performed on 16 doubles.

We measured the wall-clock time per-thread for one execution of the respective workload as well as the malloc and free times as outlined in listing 4. The colored parts are measured by enclosing them with `MPI_Wtime()`. We did perform one warmup iteration for Reduce and Broadcast but no repetitions within a run. We did not do a warmup iteration for the All-to-All workload.

These per-thread measurements are then aggregated by taking the median over all threads. We then aggregate the multiple runs by taking the 5^{th} percentile, 95^{th} percentile, and the median as shown in figure 5. We also calculated the median-based speedup relative to the baseline, which is the standard, NVIDIA provided, CUDA malloc.

```
a = malloc(n_elems*sizeof(double))
b = malloc(n_elems*sizeof(double))
// Initialize local
MPI_{Reduce, Broadcast}(a, b) // warmup
MPI_{Reduce, Broadcast, Alltoall}(a, b)
// Verify results
free(a)
free(b)
```

Listing 4. Benchmark pseudocode

Results. We observe insignificant differences for the Reduce and Broadcast workloads. Our allocators perform very similar and tend to be slightly slower than the baseline for Reduce, but slightly faster for Broadcast. However, for the more memory-heavy All-to-All operation we observe a significant speedup for both of our implementations of above 4x.

	percentile	reduce	broadcast	all-to-all
baseline	5-th	4.85	4.93	17953.02
	median	5.08	5.33	18316.86
	95-th	6.02	5.76	19527.91
no-header	5-th	4.94	4.79	3904.17
	median	5.34	5.09	4149.39
	95-th	37.97	5.61	4312.30
	speedup	0.95	1.05	**4.41**
header	5-th	5.03	4.92	3636.10
	median	5.45	5.21	4172.95
	95-th	5.78	5.755	4296.39
	speedup	0.93	1.02	4.39

Fig. 5. MPI benchmark results (values in milliseconds)

Malloc and free times are omitted, as no additional insights were collected in addition to the unit benchmarks'.

5. CONCLUSIONS

We improved in-kernel memory allocation using static and dynamic coalescing of malloc calls on GPUs, significantly accelerating programs compiled with GPU MPI.

The warp-level allocators generally outperformed the block-level variants at the expense of slightly increased allocation and de-allocation times. This trade-off is justified, as we are targeting long running applications that perform significantly more memory accesses than allocations. Nevertheless, our slowest allocating variant still significantly outperforms the NVIDIA provided base version, while only experiencing insignificant losses with respect to absolute de-allocation times.

Our best performing, warp-level allocators outperform the base version by 2-3x for most configurations of the unit benchmarks and over 4x for the MPI benchmarks, with respect to workload execution times.

In conclusion, we have presented a well-performing, generally applicable allocator that performs dynamic coalescing, as well as a static variant that relies on static analysis to coalesce memory regions. However, we believe that further improvements to the static analysis, e.g., through a comprehensive data flow analysis, would allow for more static coalescing and enable additional speedups.

6. REFERENCES

[1] Andrei Ivanov, "Gpu mpi," online: https://spclgitlab.ethz.ch/anivanov/gpu_mpi, Accessed: 2021-10-24.

[2] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg, "Scatteralloc: Massively par-

allel dynamic memory allocation for the gpu," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[3] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele, "Fast dynamic memory allocator for massively parallel architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, New York, NY, USA, 2013, GPGPU-6, p. 120–126, Association for Computing Machinery.

[4] Isaac Gelado and Michael Garland, "Throughput-oriented gpu memory allocation," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2019, PPoPP '19, p. 27–37, Association for Computing Machinery.

[5] Nvidia Corporation, "Cuda c++ best practices guide," online: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, Accessed: 2022-01-11.