Memory Defragmentation on a GPU

{veldema, philippsen}@cs.fau.de

Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl Informatik 2 Programmiersysteme

Contribution

- Program + GC + Memory Defragmentation run on the GPU
- Careful decisions as to what to run in parallel, what sequentially

Cuda++

```
//@gpu
class Test {
static Data[] d = new Data[1024];
void foo() {
if (random(100) > 50)
new Data();
else
keeper [ kernel_id() ] = new Data();
```

Definitions

Defragmentation = Compaction + Coalescing
Compaction = move objects to low addresses
Coalescing = merge free memory holes

Compaction

1 partition memory into 512K regions
2 if a region is less than 75% full with small objects, we defragment it
3 pair regions into source (low-address) and destination (high-address)
4 for each object in the source-region, move it to the destination region

   allocator/gc maintains an 'is-object-start' bit per 64 byte of memory, so we can start a gpu-thread per potential object
   use a number of free-lists per destination region to reduce list contention (lots of atomic ops to implement the lock-free free-list).

5 rewrite all pointers to point to the new object locations

   GC's mark phase records which regions contain pointers to which other regions so that we can at 512K granularity skip regions of memory unrelated to the source-regions.

Optimizations

- rewrite all elements of large arrays in parallel (better load-balancing on the GPU)
- if a 512K region is empty (maximally full free-lists for that region), coalesce it in one step by resetting the free-lists.

References

[1] Veldema, R.; Philippsen, M.: Data parallel mark&sweep on a GPU In *ISMM'11*

[2] *Dotzler, G.; Veldema, R.; Klemm, M.: JCudaMP OpenMP/Java on CUDA. In: Pankratius, V.; Philippsen, M. In* Proc. Workshop on Multicore Software Engineering (IWMSE10)

[3] Klemm, M.; Bezold, M.; Gabriel, S.; Veldema, R.; Philippsen, M. Reparallelization Techniques for Migrating OpenMP Codes in Computational Grids. In *Concurrency and Computation: Practice and Experience 21 (2009)*

Coalescing

1: find those free-memory holes that are coalescable

2: non-coalescable holes with a coalescable neighbor are the 'roots'

3: coalesce all roots in parallel

4: rebuild all free lists, skipping the holes previously marked coalescable

Lessons learned:

- do not in parallel add/remove elements from a list, better to sequentially re-create it.

What went in parallel?

- initial analysis of all objects in a region to determine if a region needs defragmentation.
- all objects moved in parallel during compaction.
- the free-lists of all regions are coalesced in parallel.
- all elements of large arrays retargeted in parallel during compaction

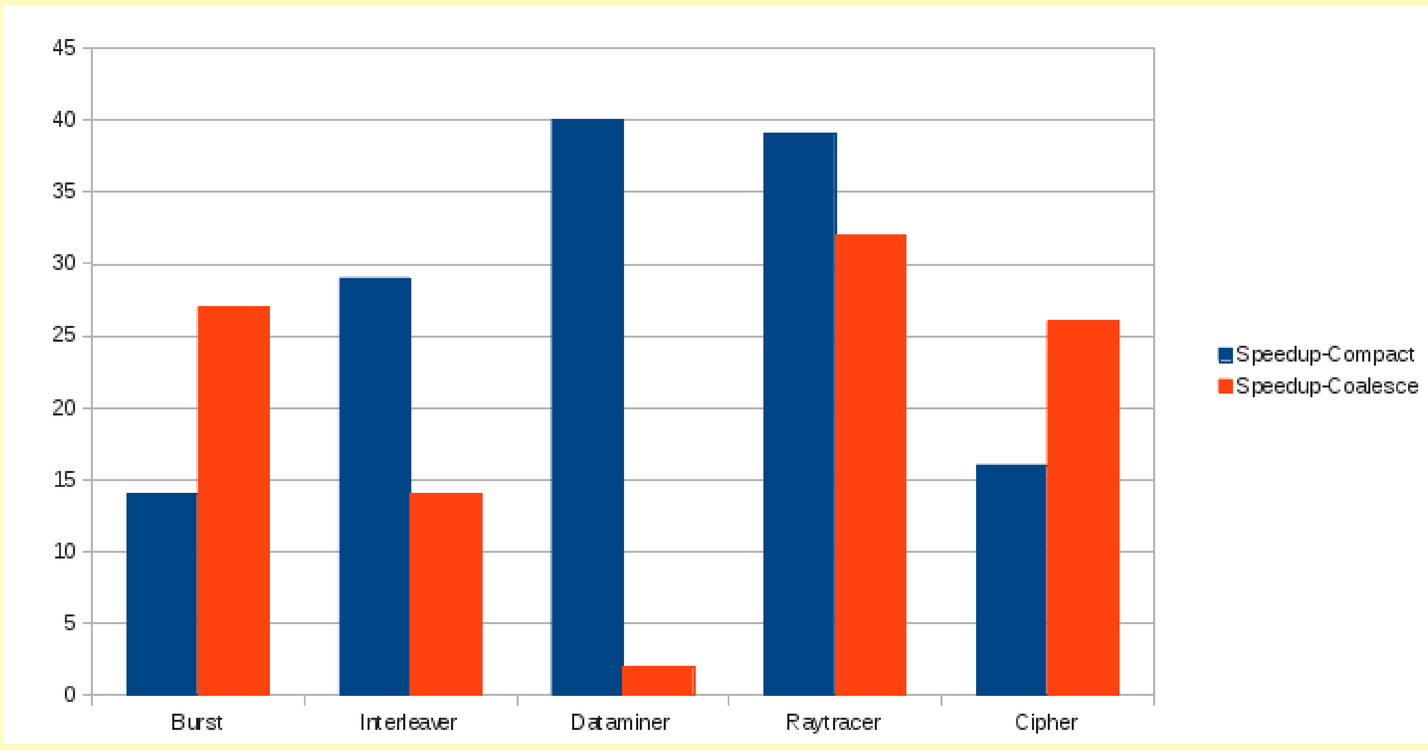
Sequential:

- finding what is coalescable with what
- rebuilding single free lists (single linked lists)

Results

Performance:

- Cuda 4.0
- GeForce 560 (384 CUDA cores)



Speedup on 384 × because:

- divergence
- atomic operations to allocate from free lists
- limited bandwidth (too few GPU-threads to overlap latencies)
- limited cache

Benchmark	defragmentation passes	compacted objects	coalesced objects	fast region coalesced	% retarget regions skipped
Burst	8	0	4088	4096	100.0
Interleaver	8	16143	95628	5545	97.8
Dataminer	3	13085	35041	4188	96.7
Raytracer	2	130902	14227	6511	98.3
Cipher	2	0	827	768	100.0

Future Work

Lots to do:

- Currently, the kernels executed on a device are not specialized/optimized. Could use CUDA's special, fast, shared memory. Could use blocking to scan arrays during pointer retargeting, etc.
- Device specific optimization as to blocking, memory-hierarchy, etc. would be useful.
- Coalescaing performance breaks down when a region is *almost* empty (large linked list). Can do better by using smaller partitions.
- Compaction speedup is low when a region is very sparsely populated. (not enough to do).
- Compaction only compacts 64 byte objects for now (arbitrary decision). Could compact larger objects too (when is it too much?)
- newer GPUs allow sharing memory between GPU and CPU transparently, how to handle a compacting GC then?

Website

`http://www2.cs.fau.de/research/JavaOpenMP/index.html`