Compacted CPU/GPU Data Compression via Modified Virtual Address Translation

LARRY SEILER, Facebook Reality Labs
DAQI LIN, University of Utah
CEM YUKSEL, University of Utah

We propose a method to reduce the footprint of compressed data by using modified virtual address translation to permit random access to the data. This extends our prior work on using page translation to perform automatic decompression and deswizzling upon accesses to fixed rate lossy or lossless compressed data.

Our compaction method allows a virtual address space the size of the uncompressed data to be used to efficiently access variable-size blocks of compressed data. Compression and decompression take place between the first and second level caches, which allows fast access to uncompressed data in the first level cache and provides data compaction at all other levels of the memory hierarchy. This improves performance and reduces power relative to compressed but uncompacted data.

An important property of our method is that compression, decompression, and reallocation are automatically managed by the new hardware without operating system intervention and without storing compression data in the page tables. As a result, although some changes are required in the page manager, it does not need to know the specific compression algorithm and can use a single memory allocation unit size.

We tested our method with two sample CPU algorithms. When performing depth buffer occlusion tests, our method reduces the memory footprint by 3.1x. When rendering into textures, our method reduces the footprint by 1.69x before rendering and 1.63x after. In both cases, the power and cycle time are better than for uncompacted compressed data, and significantly better than for accessing uncompressed data.

CCS Concepts: • **Computing methodologies** → **Image compression**; **Graphics processors**; • **Computer systems organization** → **Processors and memory architectures**.

Additional Key Words and Phrases: Shared virtual memory, lossless compression, address swizzling, page tables, tiled resources, sparse textures

ACM Reference Format:
Larry Seiler, Daqi Lin, and Cem Yuksel. 2020. Compacted CPU/GPU Data Compression via Modified Virtual Address Translation. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 19 (August 2020), 18 pages. https://doi.org/10.1145/3406177

1 INTRODUCTION

Data compression is an important technique to reduce memory bandwidth. It is extensively used in GPUs and can significantly reduce CPU memory bandwidth as well. Ideally, data compression would be combined with *compaction*, to also reduce the footprint of data in memory. Compaction is commonly used for streaming transmission and off-line storage [Le Gall 1991; Walls and MacInnis 2016], as well as to optimize page swapping in both Windows and Linux [Freedman 2000]. But with these methods, the data is decompressed before being accessed by CPU programs, due to the complication of allowing random access to compacted variable-rate compressed data.

Authors' addresses: Larry Seiler, Facebook Reality Labs; Daqi Lin, University of Utah; Cem Yuksel, University of Utah.

Compression methods that allow random access typically store the data in fixed size blocks of memory, each of which is compressed independently. This allows simple array-based addressing of the compression blocks. For lossy compression this imposes variable quality per block to meet the fixed compression rate [Iourcha et al. 1999]. For lossless compression this requires allocating the same footprint as needed for uncompressed data and also storing meta-data that specifies the type of compression performed per block. Lossless compression is standard in modern GPUs [ARM 2017; Brennan 2016] and has also been proposed for CPUs [Kanakagiri et al. 2017; Young et al. 2018].

We introduce a block-based compression method that supports random access while compacting compressed blocks into a small footprint. This builds on our earlier work providing automatic fixed rate compression and decompression by taking advantage of a new method for virtual address translation [Seiler et al. 2020]. This method is suitable for both CPU and GPU implementation. In integrated systems where CPUs and GPUs access the same memory, it allows both to access the same data using shared virtual memory [Intel 2016]. Performance testing shows that our compaction method is also beneficial for CPU-only use and we expect a similar benefit for GPU-only use.

Our method requires minimal involvement by the operating system page manager. In particular, the operating system can allocate memory in a single size and does not need to know specifics of the compression algorithm. As a result, a single operating system version can support a variety of hardware-specific CPU and GPU compression methods.

Our system:

(1) Allows CPU software to access compacted compressed data as if it is stored uncompressed,
(2) Reduces the memory footprint with better performance than our prior method,
(3) Supports writes to compacted data that increase the compressed size of data blocks,
(4) Allows CPU support for dedicated GPU compression algorithms that use compaction,
(5) Fully supports cache coherent shared virtual memory between multiple CPUs and GPUs,
(6) Allows hardware-specific compression methods to be selected per memory resource,
(7) Allows compressed resource management in app/driver code instead of kernel code, and
(8) Adds no memory latency for accessing data in regular (i.e. uncompressed) memory resources.

The next section below describes previous work relevant to accessing compacted compressed data. The remainder of the paper describes our method. First we describe how we reference compacted data in the page table (Section 3), how we manage caching & recompression (Section 4), and how compacted memory resources interact with the paging system (Section 5). Then we present performance simulations for two test cases (Section 6). Finally we discuss possible future directions for this work (Section 7).

2 BACKGROUND AND PRIOR ART

Supporting CPU random access to compacted compressed data requires changes to address translation and memory management. Address translation must map from an address range the size of the uncompressed data to variable sized blocks of compressed data. Memory management must be able to handle cases where the compression blocks get smaller or larger after writes. First we describe three previous proposals for accessing compacted data from CPUs. Then we describe aspects of our prior method for using page translation for fixed-rate compressed data [Seiler et al. 2020].

2.1 Compacted Compression via CPU Address Translation

We review several methods that have been proposed for using address translation to support random access to compacted variable rate block compressed data on CPUs [Ekman and Stenstrom 2005; Kanakagiri et al. 2017; Pekhimenko et al. 2013]. Each makes different choices for how to solve the problems involved.

A major problem is the extent of changes required in the operating system memory manager. Kanakagiri et al. [2017] avoids this complication by implementing a separate layer of address translation and memory management in the memory controller. Eliminating dependencies between the operating system and the compression system has significant advantages, but providing a second address translation and memory management layer adds a great deal of complexity.

If we accept changes to the operating system, compacted compression can be specified using the existing virtual address translation. Ekman and Stenstrom [2005] implement a three-level hierarchical scheme to manage compression data, including size fields (meta-data) in the page table along with pointers to the variable size compression blocks. Pekhimenko et al. [2013] implements a much simpler system that reduces the amount of size information required in the page table to a single size per page by compressing all cache lines within a page by the same amount. Both require support for multiple physical page sizes and both store decompressed data in the L2 cache.

Storing block size information in the virtual pages [Ekman and Stenstrom 2005] forces TLB entries to be invalidated when the compression size changes, which can be a costly operation if TLBs must be synchronized across multiple CPU and/or GPU cores. Limiting per-block size variation [Pekhimenko et al. 2013] reduces the amount of compression achievable. The alternative is to store meta-data in a separate array, as is done on modern GPUs [ARM 2017; Brennan 2016]. Young et al. [2018] notes that this works best when accesses exhibit good spatial locality, but spatial locality is required for block compression to be effective unless the blocks are very small.

The final problem involves supporting multiple physical page sizes that each require a free memory list. This requires the OS memory manager to migrate and coalesce smaller pages to create larger pages when memory becomes fragmented [Ganapathy and Schimmel 1998; Khalidi et al. 1993]. Memory management is simpler if large pages are treated as a special case [Ausavarungnirun et al. 2018]. Linux implementations exist for multiple free lists, but although Windows supports multiple large page sizes, it only supports a free list for 4KB pages [Microsoft 2018].

2.2 Automatic Decompression via Page Translation

Our method for supporting compacted compression extends our previous work that supports fixed-rate lossless and lossy compressed data using a new page translation system [Seiler et al. 2020]. Figure 1 shows how it maps a 64-bit virtual address to an offset into a 64KB physical page. It uses a four level tree of 64KB page tables, each of which holds 4K 128-bit PTEs.

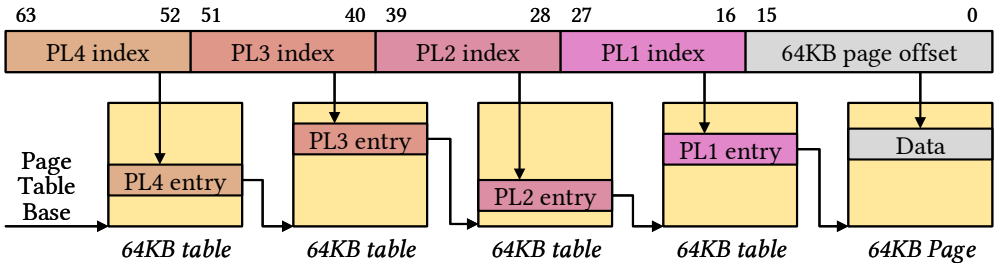


Fig. 1. *The page translation system of Seiler et al. [2020] maps 64KB physical pages.*

Existing CPUs and GPUs support a paging system that is similar except for mapping a 48-bit virtual address using 4KB pages of 64-bit PTEs. As a result, the primary change required in the operating system memory manager is to support managing 64KB aligned blocks of physical memory in place of (or in addition to) 4KB pages. Linux already supports this. Windows only fully supports memory management on 4KB physical pages, but allocates virtual addresses in aligned 64KB chunks. Therefore, the operating system change required to go from 4KB to 64KB pages is minimal.

Seiler et al. [2020] also describes an address translation mode that divides a 64KB virtual page into *subpages*, where the subpage size is a power of two from 16B to 32KB. Figure 2 illustrates the subpage mapping, where bit n determines the subpage size. PL3 (page level 3) to PL1 perform the 64KB mapping and PL0 performs the subpage mapping. A field in the PL1 PTE selects whether to use subpages and what size to use. Allowed sizes are 16B (n = 4) to 32KB (n = 15). The size of the PL0 table varies from 64KB (n = 4) to 32B (n = 15). Subpages can be used to allow support for multiple physical page sizes or to allow page properties to be specified for subpages within a 64KB page. This allows operating systems to provide full backward compatibility for software or hardware that assumes a 4KB page size.

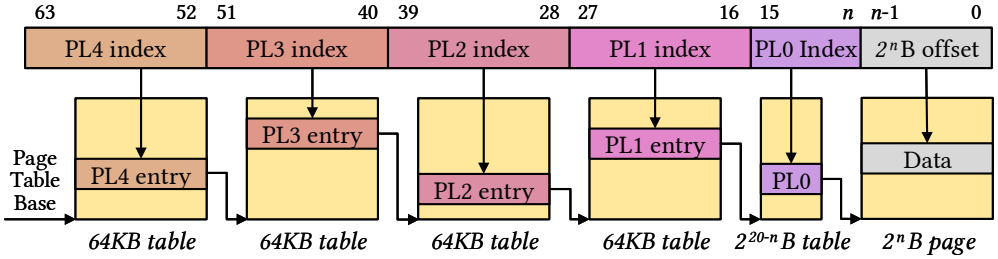


Fig. 2. *Alternate page translation system of Seiler et al. [2020] that uses the PL1 page table entry to select a PL0 table that selects smaller subpages. The subpage size can be from 16B to 32KB.*

A key feature of this new page translation is that its 128-bit PTEs store both a *Page Frame* field that selects a 64KB page and a *Subpage Frame* field that selects a 16B to 32KB subpage or a second 64KB page. Figure 3 illustrates how Seiler et al. [2020] uses page and subpage references in the PL1 PTE to support fixed rate lossless and lossy compression.

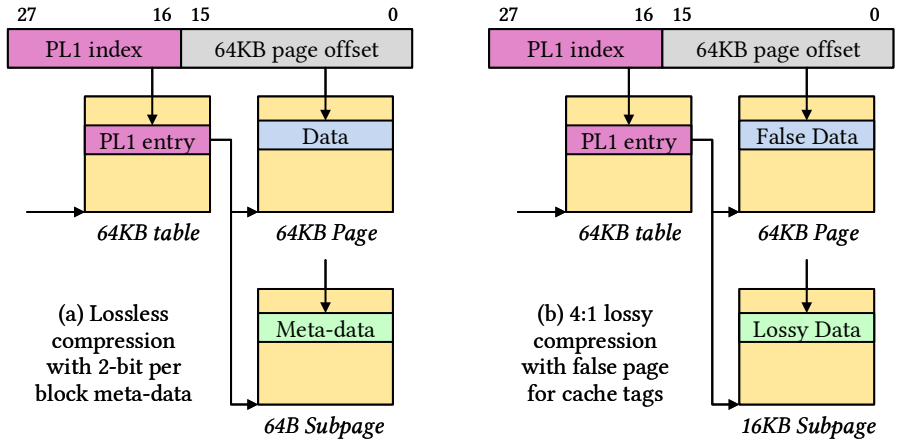


Fig. 3. *The page translation system of Seiler et al. [2020] supports both page and subpage references in the PL1 PTE: (a) shows fixed size lossless compression with a 64KB page of compression data and a 64B subpage of meta-data; (b) shows lossy compression with a 16kB (4:1 compressed) subpage and a 64KB false page (not mapped to memory) to provide L1 physical address tags.*

For lossless compression (Figure 3a), the Page Frame selects 64KB of compression blocks and the Subpage Frame selects meta-data bits that specify the compression used for each block. For

example, if the compression block size is 256B and there are 2-bits per block of meta-data, then 64B covers the 256 × 2 bits = 512 bits of meta-data corresponding to each 256 × 256B = 64KB data page.

For lossy compression (Figure 3b) the Subpage Frame selects a block of compressed data that expands to 64KB on decompression. E.g. for 4:1 lossy compression the subpage size is 16KB. The Page Frame selects a 64KB *false page*, which is not mapped to memory and is used to provide physical address tags for the uncompressed data in the L1 cache.

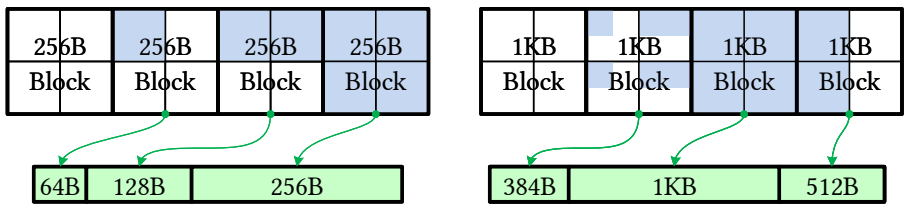
3 COMPACTING COMPRESSED DATA BLOCKS

We begin by describing how our new method stores compacted compressed data in memory. Our method supports any compression scheme that uses a power of two for the compression block size and the number of meta-data bits per block. The resulting compressed data need not be a power of 2 in size. The examples in this paper use the 256B lossless block compression method described in Seiler et al. [2020], with 2-bits of meta-data per block and a maximum of 4*x* compression. The principles described here apply equally well to other lossy or lossless variable rate compression methods.

3.1 Storing Compacted Compression Blocks

A key requirement is to support random access to the compressed data. This is simple when all of the compression blocks are the same size. When the blocks have variable sizes and are compacted to eliminate gaps, this requires a separate page translation for each block of compressed data.

Figure 4a illustrates compaction using the 256B block lossless compression scheme described in Seiler et al. [2020]. Each 256B block is compressed to 0B, 64B, 128B or 256B (the zero byte case fills the block with a clear value). The compressed data sizes are all multiples of 64B, so each of the compacted blocks can start on any 64B-aligned address. Each block requires a separate subpage pointer to where that data is stored in physical memory. As a result, each 256B block requires its own PL0 page table entry. This results in a lot of address translation overhead.



(a) Compressed 256B Data Blocks *(b) Compressed 1KB Data Blocks*

Fig. 4. *The green arrows show how our method maps compressed blocks to memory. (a) shows four 256B blocks compacted 0:4, 1:4, 2:4, and 4:4. Each block is mapped into memory at 64B alignment with no gaps in between. (b) shows four 1KB blocks that are each a 2x2 array of 256B blocks. Individual blocks are compacted as in (a) and the Subpage Frame specifies where each 1KB block is mapped in memory, again at 64B alignment with no gaps in between.*

A more efficient scheme is shown in figure Figure 4b. Here, a 2x2 set of 256B blocks are packed together into a single compacted 1KB data block. This cuts the number of PL0 page table entries required by a factor of 4. However, it also means that to know the position of the data for three of the 256B blocks in each 2x2, it is necessary to use the per-block meta-data to compute the size of the previous 256B blocks. This is an inexpensive operation. Later sections discuss other implications of sharing one subpage mapping among four 256B compression blocks.

3.2 Page Table Layout for Compacted Compression

Compacting the compression blocks requires accessing more page references during page translation than is needed for the fixed-rate compression illustrated in Figure 3. Our solution is to use the PL2 page table entry to reference two PL1 page tables, as illustrated in Figure 5.

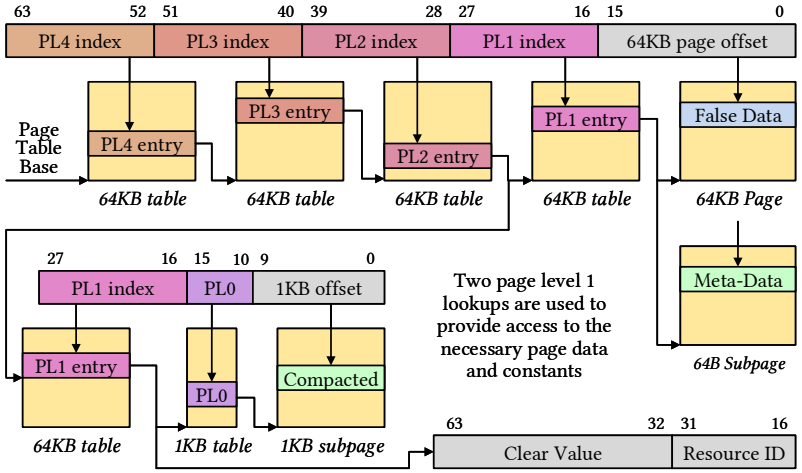


Fig. 5. *Page walk for compacted compression. The page level 2 PTE references two PL1 tables. The first PL1 table provides access to meta-data and the false data address ranges. The second PL1 table provides access to compacted data and stores per-page state in the Page Frame field. The 1KB subpage with compacted data is an unaligned region within a 64KB page (see text).*

The following data is accessed during page translation for compacted data:

- False data: provides physical address cache tags for decompressed data, in the same way as for fixed size block lossy compression.
- Meta-data: specifies the compression ratio for each 256B block, in the same way as for uncompacted lossless compression.
- Compacted data: this is a zero to 1KB region of memory that stores the compression data. It can start at any 64B aligned address and is variable length.
- Clear value: this is stored in the Page Frame field of the second PL1 table entry and specifies the clear value to use for 4:0 compressed blocks in this physical page.
- Resource ID: also stored in the PL1 table entry's Page Frame field. This specifies which compacted resource these pages are part of (see Section 4 below).

In place of the false data page, Pekhimenko et al. [2013] tags caches with the compacted physical address plus extra disambiguation bits. However, the number of disambiguation bits required depends on the maximum compression ratio that is supported, so this method fails when a block can be compressed to zero bytes. Another consideration is that CPU utility code and some GPU blocks may access compressed data directly. Therefore, the physical address of the compressed data must be tagged separately from the decompressed version of the same data.

Finally, note that each subpage of compacted data must be entirely within a single physical page, so that page remapping always treats the compacted data as one contiguous unit. With 64KB physical pages, the amount of unusable memory at the end of a page is minor. The size of the compacted data and its starting address are specified in the PL0 PTE.

4 MANAGING COMPACTED MEMORY BLOCKS

Getting good performance requires that all latency-critical operations be performed in hardware. Operating system calls are notoriously slow, primarily because security requires transitioning from user space code to protected kernel mode code whenever remapping physical memory. This section describes how compression, decompression and reallocation of data blocks is performed in hardware, without operating system intervention.

4.1 Compression Memory Path

In Seiler et al. [2020], compression and decompression occur between the L1 and L2 caches, so that the L1 cache stores decompressed data, while all other caches and memory store compressed data. Unlike that prior work, compacted compression requires being able to access compression blocks that are aligned at arbitrary multiples of the smallest compression unit size. For our example 256B block compression method, this means that compression blocks may be stored at arbitrary 64B alignments, as illustrated in Figure 4. This is simple if the L2 cache width matches the alignment, and is still relatively simple if the L2 cache is wider than the compressed data block alignment size.

Storing compressed data into the L2 cache raises an additional issue. Since four 256B blocks are stored without gaps between them, changing the size of one of the 256B blocks may require moving the others to different locations in memory. As a result, the L2 cache must store complete 2x2 blocks of compression data so that data can be moved within the cache if one of the blocks changes size. Otherwise, flushing data to the L2 cache could require a memory read to complete. As a result, our method works best for data structures that have a lot of access locality, so that the other 256B blocks in the 2x2 are likely to be used.

4.2 Recompression In Place

A key issue for compacted compression is how to deal with writes that force recompression. When a dirty compression block is evicted from the L1 cache, it must be recompressed to be stored in the L2 cache. Figure 6 illustrates two cases. The upper green memory words show the compacted data before the write. The lower green memory words show the recompacted data after the write.

Figure 6a illustrates the case where one of the blocks decreases in size. In this example, the third 1KB storage block compresses to 64B instead of 128B, as indicated by the black cross hatching. The fourth block is moved over to start immediately after the third block, leaving 64B unused at the end of the compression data.

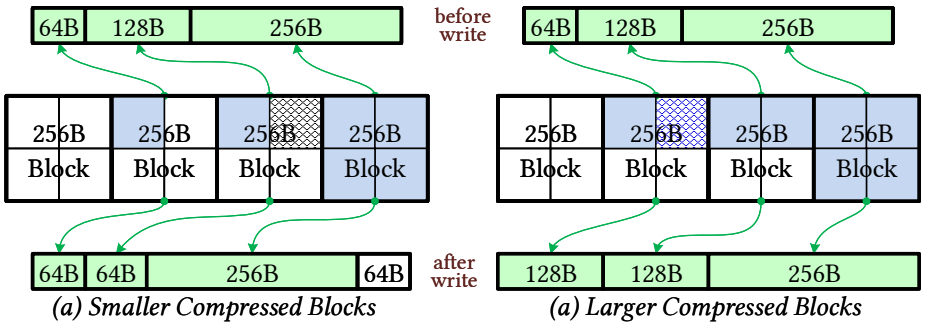


Fig. 6. *Cases for recompressing 1KB storage blocks on write back from L1 cache: (a) the third 256B block compresses to a smaller size after the write (indicated by black cross hatching), leaving a 64B gap in the stored data, (b) the second 256B block compresses to a larger size after the write (indicated by the blue cross hatching).*

It would be possible to keep track of such gaps for later reuse, but doing that would add significant complication. Instead, our method records the size of the compression block in the PL0 PTE. With compression sizes of 0, 1, 2, or 4 64B units, the possible sizes for the 1KB block are 0..14 and 16 64B units, so this field can be encoded in 4-bits. As a result, the 64B gap in Figure 6a remains part of the compression block and can be used at a later time if the compression data expands again.

4.3 Reallocating Compacted Blocks

Figure 6b illustrates the case of a 256B block increasing in size, as illustrated by the dark blue cross hatching in the second block. As a result, the second 256B block expands from 64B to 128B and the compression data no longer fits in the space allocated for it.

When the recompressed block has a larger size, new memory must be allocated to hold the data. Further, this operation must be performed in hardware without operating system intervention. This is important for performance, but more fundamentally, it may be impossible to switch tasks if dirty data in the L1 cache cannot be flushed to memory without operating system intervention.

A solution to these requirements is to allocate an extra 64KB page for each writable compacted memory resource. Then hardware can reallocate expanded compression blocks into this extra page. Doing so requires changing the page table entry and flushing the corresponding TLB entries.

When the free space on this overflow page passes some defined threshold, the hardware must signal the operating system (like a page fault, but without waiting), so that there is time for the operating system to allocate another overflow page. If a compacted memory read could result in a recompression write that doesn't fit in the overflow page, then the hardware must declare a page fault and wait for a new overflow page to be allocated. This way, it is always possible to flush the L1 cache without operating system intervention.

Ideally, each memory resource (e.g. separate array of compacted data) would have its own independent overflow page, so that different memory resources do not mix accesses on the same 64KB page. This could improve cache efficiency within a given application. More importantly, it allows different memory resources to be page managed separately, as described in Section 5.

To achieve this, a compacted page's Resource ID is used to index into an array of per-resource data that is managed by the operating system. The per-resource data structure would store the current overflow page, the current offset within that page, and the next overflow page, if one has been allocated. The operating system would manage the page addresses in the data structure and the hardware would access it and modify it when blocks are reallocated, without operating system intervention. The address and bounds of this resource array are part of the per-process state.

Finally, there is the question of what to do with the gaps left when a block is recompressed to a different location. The simplest solution is to leave the gaps in place. E.g. if an application performs only a small number of writes, very few gaps will be present. More complex solutions could be considered, where the per-resource data structure contains lists of gaps that are sorted by size. Then the hardware could select an existing gap when recompressing to a new location.

5 MANAGING COMPACTED MEMORY RESOURCES

Using our method, the operating system page manager is not involved in compressing or decompressing blocks of data. However, it does need to provide some operations to support compacted memory resources. These operations are independent of the specific compression and compaction methods that are used, so that new compression and compaction methods may be introduced without requiring an operating system update.

The following subsections cover issues related to the operating system page manager and the ways that applications may interact with it when using compressed or compacted pages. It is essential that there be no change in the complexity or efficiency of managing uncompressed pages.

5.1 Initialization and Gap Removal

At present, when an application wants to create a memory resource in virtual memory, it specifies how much memory it wants and what the page protections should be, e.g. whether writes are enabled. The operating system then sets up the necessary page tables for the desired range of memory. Although the current page size is 4KB, Windows allocates virtual addresses in aligned units of 64KB [Chen 2003] and Linux can do so as well.

The change applications must make to support compacted compression is to allocate compacted memory resources with separate operating system calls from those to allocate the rest of their virtual memory. Setting up a compacted virtual memory range requires the application to specify the array size and a compression method, which determines a small number of additional parameters:

- Compression block size and alignment
- Compression method identifier to store in the Select field
- Meta-data size and the value that specifies using the clear color
- Clear color and a generated Resource ID

Given these parameters, the page manager sets up a virtual address range and initializes the per-resource data structure. Initially, all compression blocks are set to use the clear color, so that all of the compacted memory blocks are stored in zero bytes. Virtual address ranges for compressed (not compacted) memory resources can be allocated by a similar process, minus some of the parameters.

The next step is to load data into the compacted memory resource, if the clear color isn't the desired initial value. For efficiency, this should be done a compaction block at a time. The result is that when each compaction block is written, it is reallocated into the overflow page. But no gap is left by this reallocation because the initial clear color compaction blocks use no bytes of memory. So a compacted memory resource may be created and initialized without introducing any gaps.

The same method may be used to remove gaps embedded in an existing compacted resource. The application allocates a new memory resource that is initialized to the clear color. Then the application copies the existing compacted resource into it, a compaction block at a time. Finally, the old memory resource is deallocated.

5.2 Compressed Page Management

Our prior work [Seiler et al. 2020] does not consider the question of how the operating system manages shared pages. Even with fixed-size uncompacted compression, multiple PL1 PTEs can reference the same page of meta-data or the same page of lossy compressed data. This is an issue because of how operating systems manage pages.

Operating system page managers typically contain a *Frame Table*, which is also called a *Page Frame Number Database*. This table specifies information about each physical page, including statistics, its current state, and which process's virtual page maps to this physical page, if any. This data is used when remapping virtual pages to physical pages [Russinovich and Solomon 2004].

For uncompressed data, there are normally zero or one virtual addresses mapped to each physical page. Mapping two or more virtual addresses to the same page is referred to as address aliasing. Aliasing is used for specialized purposes, e.g. to allow one process to write data into memory that is read by another process. Address aliasing is typically a rare event that is costly to support in the page manager, since the Frame Table itself can only specify one virtual page per physical page.

Compressed pages as per Seiler et al. [2020] map multiple virtual pages to each meta-data page or lossy compressed page. This does not alias memory, since virtual pages of the same process access disjoint subpages of the shared physical page. E.g., with 2-bits of meta-data per 256B lossless block, up to 1K virtual pages can each access their own 64B out of a shared meta-data page. Still, in terms of the Page Frame, the page addresses are aliased.

A simple solution to the problem (not mentioned in Seiler et al. [2020]) is to mark the Frame Table as storing compressed data and point it to the first virtual address that maps to that page. All other virtual pages that map to the same physical page will use successively larger virtual addresses. Therefore the complete set of virtual addresses that map to subpages of the same physical page can be easily discovered by stepping through the page table and checking the physical page mappings.

5.3 Compacted Page Management

With our compacted compression method, reallocated blocks can result in non-contiguous virtual addresses being mapped to the same physical page. So the page manager can't store a single virtual address in the Frame Table and directly use that to find the other virtual addresses that reference the same physical page.

The Resource ID provides a solution to the problem. In our method, each 64KB page of compacted memory specifies a 16-bit resource ID that allows the hardware to look up memory resource dependent information such as the overflow page(s) for reallocating compacted blocks. The virtual address of the array that the Resource ID indexes to find this information is the only additional machine state that must be stored for the process on a context switch.

The information indexed by the Resource ID can also include the virtual address range of the resource. For compacted pages, the page manager can store in the Page Frame any virtual address that references that physical page. Then the memory manager can use that virtual address to read the Resource ID and use it to look up the virtual address range of the resource, which includes all of the virtual addresses that may reference that physical page.

A simple though coarse grain way for the page manager to use the Resource ID is to deallocate all physical pages mapped to that memory resource. This is useful e.g. when the application is done using a particular memory resource. Another alternative is for the application to use multiple Resource IDs for a single logical resource. Then the logical resource can be deallocated in sections.

Another simple alternative is for the hardware to identify when a compacted page contains non-contiguous virtual addresses. This occurs when the overflow page is used, other than during a sequential initialization operation. In that case, a bit could be set in the Frame Table entry that indicates that the physical page has discontiguous virtual addresses mapped to it. If there aren't many writes to the compacted memory resource, then relatively few pages will be discontiguous. If a page is discontiguous, the page manager could search, starting at the specified virtual address, to identify pages with the same Resource ID that map to that physical page. If the bit is clear, the simple method defined above for compressed resources could be used. More complex mechanisms could also be defined if usage requires them, without affecting the hardware implementation.

Finally, we need to consider the impact of an operating system call on the compression and compaction hardware. Calling the operating system kernel normally results in flushing the translation lookaside buffer (TLB) that is used to optimize address translation. For our method, the TLB also stores information on how to recompress data that is stored in the L1 cache. So a kernel call will flush all decompressed data from the L1 cache to the L2 cache. However, it will not have any other impact on the compression hardware beyond its impact on noncompressed data.

6 PERFORMANCE EVALUATION

Our work provides a way to enable compacted storage of variable-rate compressed data and allows that data to be shared between CPUs and GPUs. The key goal is to reduce the footprint of the compressed data in memory, without increasing power or cycle time relative to using compressed data without compaction. This can make both CPU-only and GPU-only applications more efficient. It also allows applications on integrated CPU/GPU systems to perform operations on whichever processor is more power efficient for each stage. At present, this can only be done by

decompressing data when passing it from the GPU to the CPU, which makes most such integrated use cases impractical.

To evaluate whether our method achieves this goal, we created two sample applications. One performs occlusion testing on the CPU using a depth buffer and the other uses the CPU to blend pregenerated text into textures. Both the depth buffer and the textures are assumed to be generated on a GPU using lossless compression, but our tests cover only the CPU operations performed to access memory. We measure each application in three test cases:

(1) *Uncompressed* (standard case): The GPU data is decompressed prior to access by the CPU.
(2) *Compressed* ([Seiler et al. 2020]): Compressed GPU data uses the full uncompressed footprint.
(3) *Compacted* (our method): The CPU accesses compressed and compacted GPU data.

Our primary result is that compaction greatly reduces the memory footprint required to store compressed data: the memory footprint is 33% of the bytes for the occlusion test and 59% of the bytes before the text rendering test. Further, reallocating compacted blocks that no longer fit in their old location increased the footprint by just 2.4%. This validates that the simple method of reallocating to an overflow page, rather than attempting to perform gap filling, can be a viable strategy for solving the reallocation problem.

Another important result of our work is that in both tests, compaction uses less power and fewer memory access cycles than compression without compaction, in the range of about 2% to 8%. This benefit is despite the added complexity of tracking variable sized compression blocks. So for applications such as rendering that have sufficient access locality, there is no need to make a trade-off: compaction is uniformly better for power and performance, and dramatically better for memory footprint. This benefit applies to CPU-only applications, as well as for exchanging data between CPUs and GPUs. We expect it to apply to GPU-only applications as well.

6.1 Simulation Methodology

For the most part, our simulation methodology follows that reported in Seiler et al. [2020]. In particular, we use the same compression method, which performs 4:4, 4:2 or 4:1 compression but does not support a clear value. However, there are four key areas where this simulation differs.

First, we assume that the data is accessed in swizzled form. That is, each 1KB chunk of memory contains 32x16 16-bit depth values for the occlusion test, or 16x16 32-bit pixels for the rendering test. This matches the way that GPUs typically store blocks of data in order to achieve best performance, but it differs from Seiler et al. [2020], which uses row-major arrays.

Second, for each of the three methods we prefetch into the L2 cache the data required for a 1KB uncompressed block, instead of a 256B block as in Seiler et al. [2020]. This is required for our compaction method and actually improves performance for the uncompressed and compressed cases. So using this prefetching for all three cases allows a more uniform comparison.

Third, we changed the decompression latency from 50 cycles to 25 cycles. Discussions with silicon designers led us to conclude that even 25 cycles is a conservative added latency for an optimized design. We retained the assumption that only one block is decompressed at a time.

Finally, we changed from scalar processing to using SSE-128 vector instructions, which provide sixteen 128-bit vector registers. This lets CPU code issue 256B of memory reads before waiting for data to come back from cache or memory. This change was applied to all three test cases.

6.2 Memory Path Simulation

In our method, compression and decompression occur between the L1 and L2 caches, as in [Seiler et al. 2020]. The CPU we simulate is based on the standard 2010 Intel core i7 configuration, which has a clock speed of 3.2 GHz and a single channel 16GB DDR3-1600 DRAM.

The configuration of the cache system is shown in Table 1. There is no L3 cache because we tested a single-core configuration (see end of Section 7). All reads and writes are pipelined, but a simplified policy is enforced such that if a read miss happens in the L1 cache, the pipeline stalls for data to come back from the L2 cache or from the DRAM.

Table 1. *Detailed configuration of hardware components, based on the 2010 Intel core i7 CPU.*

Component	Capacity	Associativity	Access Latency	Read Energy	Write Energy
L1 Cache	32,768 B	8-way	4 cycles	0.0813 nJ	0.0816 nJ
Meta L1 Cache	32,768 B	8-way	4 cycles	0.0813 nJ	0.0816 nJ
L2 Cache	262,144 B	8-way	10 cycles	0.1802 nJ	0.1998 nJ
TLB (Level 1)	64 entries	4-way	1 cycle	0.0308 nJ	0.0289 nJ
TLB (Level 2)	512 entries	4-way	6 cycles	0.0449 nJ	0.0513 nJ

Tables in the following two sections compare the three test cases based on memory statistics, energy use, and number of clock cycles required to access memory. Columns give the energy or cycle time for categories and subcategories, the percentage of the total within each category, and the ratio of how this value compares to case (2), which is based on Seiler et al. [2020]. Cycle counts specify the number of cycles to deliver data to the next stage plus the number of cycles the next stage is stalled waiting for data. For example, the line labeled *Data L1 Read* counts one clock cycle for each L1 hit since the L1 cache is pipelined, but counts 4 clocks to restart the pipeline on a miss.

6.3 Software Occlusion Application

Our first application uses the CPU to perform software occlusion culling. This involves pre-computing whether a large number of complex but small objects, e.g. characters, are occluded by the scene background. The background objects are referred as occluders and their positions form the content of the depth buffer. The foreground objects being tested are referred as occludees. Figure 7 shows the occlusion scene that we used.

Fig. 7. *The left image shows a sample scene of the software occlusion test that contains 27,025 meshes with 476 meshes being visible. Ground, wall, and buildings are occluder in the scene. The right image shows a global overview of scene, which contains many repetitive objects (like wooden baskets, containers, and stands) but most are occluded from the ground view. Notice that objects outside are culled by the viewing frustum before culling using 2D axis aligned bounding boxes (AABBs). On average more than 10,000 2D AABBs are tested against the depth buffer.*

We adapted our test from the scene and implementation provided by Intel [Kuah 2016] to make it more efficient in our CPU/GPU environment. While the Intel implementation uses SIMD instructions to draw the depth buffer on the CPU, we assume in all three cases that the depth buffer

has already been rendered on the GPU and that the CPU can read the depth buffer directly. This is in contrast to methods that need to copy data from GPU and deswizzle and/or decompress the data before the CPU can access it.

To simplify the occlusion test, we use 2D screen-space axis aligned bounding boxes (AABBs) instead of the 3D AABBs used in the Intel implementation. Since we use 2D AABBs, we conservatively use the nearest z of the object as the depth value such that an object is only invisible if the nearest z is farther than the z value in the depth buffer. We also changed the bit depth of the depth buffer from 32 bit to 16 bit. The algorithm steps are as follows:

(1) Use the GPU to render a 16-bit depth buffer of the background objects, several frames ahead.
(2) Find the 2D bounding box of each object for which occlusion is to be tested.
(3) Read the depth buffer within the bounding box and compare to the object depth (for simplicity, we use a single conservative object depth over the entire bounding box).
(4) Schedule the object for rendering if any part of it is in front of the depth buffer.

Table 2. *Memory and cache usage statistics in the software occlusion culling test.*

	(1) Uncompressed (standard) 4KB pages		**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages		**(3) Compacted** (our method) 64KB pages, 1KB subpages	
		ratio		*ratio*		*ratio*
Lines read from memory	**10.95 M**	2.93	**3.74 M**	1.00	**2.88 M**	0.77
L1 hit rate	**87.5 %**	0.88	**99.6 %**	1.00	**99.6 %**	1.01
L2 hit rate	**94.5 %**	1.16	**81.3%**	1.00	**85.4%**	1.05
Memory footprint	**398.44 MiB**	1.00	**398.83 MiB**	1.00	**129.85 MiB**	0.33
–Meta Data Footprint	0	0.00	0.39 MiB	1.00	0.39 MiB	1.00
–Data Footprint	398.44 MiB	1.00	398.44 MiB	1.00	129.44 MiB	0.32
Valid Data Size	**398.44 MiB**	3.08	**129.44 MiB**	1.00	**129.44 MiB**	1.00

Table 2 shows that our memory compaction method reduces the memory footprint to 1/3 of that required for uncompressed data or for compressed data this is not compacted. More surprisingly, our method also reduces the lines read from memory relative to (2) by 23%. This turns out to be due to getting a better L2 hit rate, which in turn is due to compacted data packing more efficiently into the L2 cache. Also note that the L1 hit rate is higher for (2) and (3) than for (1). This is because in (1) a single cacheline is loaded into L1 on a miss, whereas for (2) and (3) all four cachelines of a compression block are loaded. This also explains why the L2 hit rate is higher in (1) than in (2).

Table 3. *Energy Breakdown of hardware components in millijoules in the software occlusion culling test, with ratios of uncompressed and compacted data to the compressed data in column (2).*

	(1) Uncompressed (standard) 4KB pages			**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages			**(3) Compacted** (our method) 64KB pages, 1KB subpages		
	energy	*%*	*ratio*	*energy*	*%*	*ratio*	*energy*	*%*	*ratio*
TLB	0.10	0.04 %	0.99	0.10	0.08 %	1.00	0.17	0.17 %	1.82
L2-TLB	0.01	0.01 %	0.44	0.03	0.03 %	1.00	0.05	0.05 %	1.82
Data L1	8.06	3.48 %	1.00	8.07	7.06 %	1.00	8.23	7.86 %	1.02
Meta L1	0.00	0.00 %	0.00	0.27	0.23 %	1.00	0.27	0.25 %	1.00
L2	35.49	15.33 %	10.74	3.30	2.89 %	1.00	3.36	3.21 %	1.02
Codec	0.00	0.00 %	0.00	6.17	5.40 %	1.00	6.20	5.92 %	1.00
Memory	187.80	81.14 %	1.95	96.37	84.31 %	1.00	86.52	82.54 %	0.90
Total (rendering)	**231.45**	**100.00 %**	2.02	**114.31**	**100.00 %**	1.00	**104.81**	**100.00 %**	0.92
Initialization	**91.45**	**(*extra*)**	∞	**0.00**	- - -	0	**0.00**	- - -	0

Table 3 shows a comparison of the energy consumption related to memory loads for the three test cases. Compression results in over 2x power savings relative to using uncompressed data. Our compaction method (3) reduces power by another 8%. This is primarily due to our method using less memory power, which in turn is due to the better L2 hit rate and reduced memory reads, as described above.

Table 4. *Time Breakdown in CPU cycles in the software occlusion culling test, with ratios of uncompressed and compacted data to the compressed data in column (2).*

	(1) Uncompressed (standard) 4KB pages			**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages			**(3) Compacted** (our method) 64KB pages, 1KB subpages		
	# of cycles	*%*	*ratio*	*# of cycles*	*%*	*ratio*	*# of cycles*	*%*	*ratio*
VA Translation	**10.91 M**	**1.90 %**	0.79	**13.85 M**	**3.47 %**	1.00	**37.05 M**	**9.51 %**	2.67
– TLB Read	3.09 M	0.54 %	0.99	3.11 M	0.78 %	1.00	5.67 M	1.45 %	1.82
– L2-TLB Read	1.77 M	0.31 %	0.44	3.99 M	1.00 %	1.00	7.25 M	1.86 %	1.82
– Data L1 Read	0.66 M	0.12 %	0.82	0.81 M	0.20 %	1.00	5.68 M	1.46 %	7.03
– L2 Read	0.67 M	0.12 %	0.70	0.95 M	0.24 %	1.00	3.40 M	0.87 %	3.57
– Memory Read	4.71 M	0.82 %	0.95	4.99 M	1.25 %	1.00	15.05 M	3.86 %	3.02
Load	**561.95 M**	**98.10 %**	1.46	**385.51 M**	**96.53 %**	1.00	**352.63 M**	**90.49 %**	0.91
– Data L1 Read	191.31 M	33.40 %	1.36	140.51 M	35.18 %	1.00	144.33 M	37.04 %	1.03
– Meta L1 Read	0.00 M	0.00 %	0.00	13.09 M	3.28 %	1.00	13.16 M	3.38 %	1.00
– L2 Read	123.07 M	21.48 %	3.61	34.08 M	8.53 %	1.00	34.24 M	8.79 %	1.00
– Memory Read	247.57 M	43.22 %	2.06	120.46 M	30.16 %	1.00	83.17 M	21.34 %	0.69
– Codec Delay	0.00 M	0.00 %	0.00	77.37 M	19.37 %	1.00	77.73 M	19.95 %	1.00
Total (rendering)	**572.86 M**	**100.00 %**	1.43	**399.36 M**	**100.00 %**	1.00	**389.68 M**	**100.00 %**	0.98
Initialization	**151.97 M**	**(*extra*)**	∞	**0**	- - -	0	**0**	- - -	0

Table 4 shows a comparison of the number of processor cycles related to memory access between for the three test cases. The VA translation time is higher in our method, but this is counteracted by reduced memory read time into the L2 cache, due to a better L2 hit rate as described above. Overall, our method is 2% faster than compressed data and using uncompressed data is 43% slower than compressed data. That is without considering the time needed to copy and decompress data from a GPU if the CPU requires uncompressed data, as shown on the bottom line of the table.

6.4 Text Rendering Application

To test our method in the context of writable data, we perform a text rendering task (Figure 8) on the same 668 game asset textures (from Zero Day [Beeple 2015], San Miguel, Crytek Sponza, and Amazon Lumberyard Bistro [McGuire 2017]) that we used in Seiler et al. [2020]. These include 5GB of albedo maps, normal maps, opacity maps, and specular color maps to provide a range of data types and compression ratios on 32-bit pixels. Figure 8 illustrates the operation.

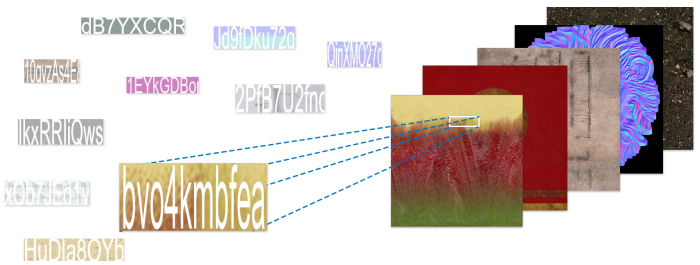


Fig. 8. *An illustration of the text rendering test, which blends 64×16 to 256×64 blocks of pregenerated random text into 668 textures that contain a wide variety of different kinds of data.*

For this test we blend 10,000 single-line prerendered anti-aliased rectangular blocks of text into random textures at random positions and measure the performance of memory operations in terms of energy consumption and processor cycles. The sizes of the text blocks range from 256×64 to 64×16. This requires reading and then writing back the texture data to combine the text block with the texture. Generating the rasterized text blocks is not included in the test, since we are measuring accesses to the compacted texture data.

Table 5. *Memory and cache usage statistics in the text rendering test, with ratios of our methods to the with ratios of uncompressed and compacted data to the compressed data in column (2).*

	(1) Uncompressed (standard) 4KB pages		**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages		**(3) Compacted** (our method) 64KB pages, 1KB subpages	
		ratio		*ratio*		*ratio*
Lines read from memory	**9.78 M**	1.49	**6.57 M**	1.00	**5.75 M**	0.88
L1 hit rate	**86.8 %**	0.91	**95.9 %**	1.00	**95.9 %**	1.00
L2 hit rate	**91.5 %**	1.27	**72.2 %**	1.00	**76.2 %**	1.06
Memory footprint	**4654.76 MiB**	1.00	**4659.31 MiB**	1.00	**2749.34 MiB**	0.59
–Meta Data Footprint	0	0.00	4.55 MiB	1.00	4.55 MiB	1.00
–Data Footprint	4654.76 MiB	1.00	4654.76 MiB	1.00	2679.33 MiB	0.58
–New Allocation	0.00 MiB	1.00	0.00 MiB	- - -	65.46 MiB	∞
Valid Data Size	**4654.76 MiB**	1.74	**2679.33 MiB**	1.00	**2679.33 MiB**	1.00

Table 5 presents memory statistics for this test. Unlike the occlusion test, this application writes data back to memory and therefore faces the reallocation problem described in Section 4.3. We tested the simple method of allocating new memory any time a 1KB storage block recompresses to a larger size. The result is that the footprint of the compacted data increases by just 2.4% as a result of running the test. Even after running the test, our compaction method saves 41% of the memory footprint required without compaction.

Table 6. *Energy Breakdown of hardware components in millijoules in the text rasterization test, with ratios of uncompressed and compacted data to the compressed data in column (2).*

	(1) Uncompressed (standard) 4KB pages			**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages			**(3) Compacted** (our method) 64KB pages, 1KB subpages		
	energy	*%*	*ratio*	*energy*	*%*	*ratio*	*energy*	*%*	*ratio*
TLB	0.00	0.00 %	1.00	0.00	0.00 %	1.00	0.02	0.02 %	4.89
L2-TLB	0.01	0.00 %	1.00	0.01	0.01 %	1.00	0.02	0.02 %	2.85
Data L1	2.63	1.87 %	1.00	2.63	2.51 %	1.00	2.69	2.68 %	1.02
Meta L1	0.00	0.00 %	0.00	0.19	0.19 %	1.00	0.22	0.22 %	1.12
L2	12.96	9.23 %	4.92	2.64	2.52 %	1.00	2.85	2.84 %	1.08
Codec	0.00	0.00 %	0.00	3.17	3.03 %	1.00	3.54	3.53 %	1.12
Memory	124.88	88.89 %	1.30	96.02	91.75 %	1.00	90.95	90.69 %	0.95
Total (rendering)	**140.49**	**100.00 %**	1.34	**104.66**	**100.00 %**	1.00	**100.29**	**100.00 %**	0.96
Initialization	**1239.65**	**(*extra*)**	∞	**0.00**	- - -	0	**0.00**	- - -	0

Table 6 and Table 7 show the energy and power used for the text rasterization test cases. The saving of energy and time of (2) and (3) over the uncompressed data is smaller than in the software occlusion test. This is expected since this test has a smaller compression ratio (1.74x vs. 3.08x). Still, the uncompressed data case (1) uses 34% more energy than the compressed case (2), and our method (3) beats that by 4%. For memory access cycles, the uncompressed case uses 12% more cycles than compressed (2) and our compacted method (3) beats that by 2%. This means that the extra cost for virtual address translation (caused by smaller page size) and the cost of allocating new space are offset by the larger savings of having a compacted representation of the data.

Table 7. *Time Breakdown in CPU cycles in the text rasterization test, with ratios of our methods to the with ratios of uncompressed and compacted data to the compressed data in column (2).*

	(1) Uncompressed (standard) 4KB pages			**(2) Compressed** [Seiler et al. 2020] 64KB pages, 4KB subpages			**(3) Compacted** (our method) 64KB pages, 1KB subpages		
	# of cycles	*%*	*ratio*	*# of cycles*	*%*	*ratio*	*# of cycles*	*%*	*ratio*
VA Translation	**13.31 M**	**5.09 %**	0.67	**19.99 M**	**8.60 %**	1.00	**30.39 M**	**13.38 %**	1.52
– TLB Read	0.13 M	0.05 %	1.00	0.13 M	0.06 %	1.00	0.65 M	0.29 %	4.89
– L2-TLB Read	0.80 M	0.31 %	1.00	0.80 M	0.34 %	1.00	2.28 M	1.00 %	2.85
– Data L1 Read	1.03 M	0.39 %	0.92	1.11 M	0.48 %	1.00	2.72 M	1.20 %	2.44
– L2 Read	0.78 M	0.30 %	0.67	1.16 M	0.50 %	1.00	1.58 M	0.70 %	1.37
– Memory Read	10.58 M	4.05 %	0.65	16.29 M	7.01 %	1.00	22.56 M	9.93 %	1.38
– Queueing Delay	0.00 M	0.00 %	0.00	0.50 M	0.21 %	1.00	0.60 M	0.27 %	1.22
Load	**229.21 M**	**87.71 %**	1.18	**193.82 M**	**83.35 %**	1.00	**178.01 M**	**78.38 %**	0.92
– Data L1 Read	42.53 M	16.27 %	1.66	25.63 M	11.02 %	1.00	25.63 M	11.29 %	1.00
– Meta L1 Read	0.00 M	0.00 %	0.00	4.83 M	2.08 %	1.00	4.83 M	2.13 %	1.00
– L2 Read	42.03 M	16.09 %	3.01	13.96 M	6.00 %	1.00	13.96 M	6.15 %	1.00
– Memory Read	143.64 M	54.97 %	1.21	118.43 M	50.93 %	1.00	109.75 M	48.32 %	0.93
– Queueing Delay	1.01 M	0.39 %	0.09	11.03 M	4.74 %	1.00	3.89 M	1.71 %	0.35
– Codec Delay	0.00 M	0.00 %	0.00	19.95 M	8.58 %	1.00	19.95 M	8.78 %	1.00
Store	**18.73 M**	**7.17 %**	1.00	**18.73 M**	**8.05 %**	1.00	**18.73 M**	**8.25 %**	1.00
Total (rendering)	**261.31 M**	**100.00 %**	1.12	**232.54 M**	**100.00 %**	1.00	**227.12 M**	**100.00 %**	0.98
Initialization	**2023.09 M**	(*extra*)	∞	**0**	- - -	0	**0**	- - -	0

7 DISCUSSION AND FUTURE WORK

Our work shows that it is practical and useful to support compacted data compression on CPUs to reduce the memory footprint, even when not sharing data between CPUs & GPUs. The footprint reduction for our two example applications range from about 1.5x to 3x, even with the simple compression scheme that we tested. Further, for the applications that we tested, performance and power modestly improve when using compaction instead of just using compression.

There are a great many kinds of data structures that can benefit from compacted compression, whether they run on CPUs, GPUs, or both. In general, these would be any data structure that allows significant compression and has sufficient locality of access to make multi-cacheline reads into the caches efficient. Some examples of benefits for data structures shared by CPUs and GPUs are:

- Vertex and index buffers: generated on the CPU with compaction and then read on the GPU.
- Streaming buffers: generated compacted on the GPU to be saved for later use.
- G-buffer data: written once and read once, so reallocation is not required.
- Multisampling: most pixels require just 1 or 2 fragment colors.
- Texture compression: allow the bit-rate per block to vary based on visual quality.

Supporting the same compacted compression methods on both CPU and GPU allows integrated systems to select the best processor for each task, without concern for data transfer or reformatting. E.g., font rasterization may be more efficient on a CPU [Hartmeier 2016] and depth buffer generation may be more efficient on a GPU [Kuah 2016].

There are a number of directions for future work with this compaction method:

- Experiment with more elaborate compression schemes, including using constant fill colors.
- Use it support existing or new page-on-demand decompression methods.
- Try using compaction with additional applications, including GPU-only applications.
- Support more efficient streaming transmission by increasing the use of compaction.
- Test configurations that include an L3 cache and run the application on multiple cores.

8 CONCLUSION

We have described a way to modify the CPU page translation system to support compacted storage of randomly addressable, variable compression rate data. Our proposed method requires a new logic block between the L1 and L2 caches that performs compression, decompression, and reallocation. This added logic has minimal impact on the CPU memory access path.

Our method allows dramatic reductions in the memory footprint needed for compressed data. Despite the complexity of tracking the locations of variable sized blocks, performance and power usage in our test cases is better than using compression without compaction, which in turn is significantly better than using uncompressed data.

Further, our method has minimal impact on the operating system page manager, which doesn't require any knowledge of the compression or decompression methods. The page manager only needs to support one page size, so that support for compaction has minimal impact on the efficiency or complexity of the page manager when processing uncompacted memory resources.

We believe that our method is practical for hardware implementation in CPUs and GPUs and for software implementation in operating system page managers. Our results show that it could provide significant footprint reduction and associated performance improvement & power reduction for combined CPU/GPU applications as well as for applications running only on CPUs or GPUs.

ACKNOWLEDGMENTS

This project was supported in part by a grant from Facebook Reality Labs.

REFERENCES

ARM. 2017. *Arm Frame Buffer Compression.* https://developer.arm.com/architectures/media-architectures/afbc

Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2018. Mosaic: Enabling Application-Transparent Support for Multiple Page Sizes in Throughput Processors. *ACM SIGOPS Operating Systems Review* 52, 1 (2018), 27–44.

Beeple. 2015. *Cinema 4D Project Files.* https://www.beeple-crap.com/resources

Chris Brennan. 2016. *Delta Color Compression Overview.* https://gpuopen.com/dcc-overview/

Raymond Chen. 2003. *Why is address space allocation granularity 64K?* https://devblogs.microsoft.com/oldnewthing/20031008-00/?p=42223

Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *ACM SIGARCH Computer Architecture News*, Vol. 33. 74–85.

Michael J Freedman. 2000. The compression cache: Virtual memory compression for handheld computers. (2000).

Narayanan Ganapathy and Curt Schimmel. 1998. General Purpose Operating System Support for Multiple Page Sizes.. In *USENIX Annual Technical Conference.* 91–104.

Martina K. Hartmeier. 2016. *Software vs. GPU Rasterization in Chromium.* https://software.intel.com/en-us/articles/software-vs-gpu-rasterization-in-chromium

Intel. 2016. *OpenCL™ 2.0 Shared Virtual Memory Overview.* https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview

Konstantine I Iourcha, Krishna S Nayak, and Zhou Hong. 1999. System and method for fixed-rate block-based image compression with inferred pixel values. US Patent 5,956,431.

Raghavendra Kanakagiri, Biswabandan Panda, and Madhu Mutyam. 2017. MBZip: Multiblock data compression. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017), 1–29.

Yousef A Khalidi, Madhusudhan Talluri, Michael N Nelson, and Dock Williams. 1993. Virtual memory support for multiple page sizes. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III.* IEEE, 104–109.

Kiefer Kuah. 2016. *Software Occlusion Culling.* https://software.intel.com/content/www/us/en/develop/articles/software-occlusion-culling.html

Didier Le Gall. 1991. MPEG: A video compression standard for multimedia applications. *Commun. ACM* 34, 4 (1991), 46–58.

Morgan McGuire. 2017. *Computer Graphics Archive.* https://casual-effects.com/data

Microsoft. 2018. *Large-Page Support.* https://docs.microsoft.com/en-us/windows/win32/memory/large-page-support

Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: a low-complexity, low-latency main memory compression framework.

In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 172–184.

Mark E Russinovich and David A Solomon. 2004. *Microsoft Windows Internals: Microsoft Windows Server (TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press.

Larry Seiler, Daqi Lin, and Cem Yuksel. 2020. Automatic GPU Data Compression and Address Swizzling for CPUs via Modified Virtual Address Translation. In *Symposium on Interactive 3D Graphics and Games (I3D 2020)*. ACM Press, New York, NY, USA, 10. https://doi.org/10.1145/3384382.3384533

Frederick G Walls and Alexander Sandy MacInnis. 2016. VESA display stream compression for television and cinema applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6, 4 (2016), 460–470.

Vinson Young, Sanjay Kariyappa, and Moinuddin K Qureshi. 2018. CRAM: Efficient Hardware-Based Memory Compression for Bandwidth Enhancement. *arXiv preprint arXiv:1807.07685* (2018).