Automatic GPU Data Compression and Address Swizzling for CPUs via Modified Virtual Address Translation

Larry Seiler
Facebook Reality Labs

Daqi Lin
University of Utah

Cem Yuksel
University of Utah

ABSTRACT

We describe how to modify hardware page translation to enable CPU software access to compressed and swizzled GPU data arrays as if they were decompressed and stored in row-major order. In a shared memory system, this allows CPU to directly access the GPU data without copying the data or losing the performance and bandwidth benefits of using compression and swizzling on the GPU.

Our method is flexible enough to support a wide variety of existing and future swizzling and compression schemes, including block-based lossless compression that requires per-block meta-data.

Providing automatic compression can improve performance, even without considering the cost of copying data. In our experiments, we observed up to 33% reduction in CPU/memory energy use and up to 35% reduction in CPU computation time.

CCS CONCEPTS

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

KEYWORDS

Shared virtual memory, lossless compression, address swizzling, page tables, tiled resources, sparse textures

ACM Reference Format:
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 '20), May 5–7, 2020, San Francisco, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3384382.3384533

1 INTRODUCTION

Data compression and address swizzling are important techniques to reduce GPU memory bandwidth and, therefore, increase performance per unit of power. GPUs have long used address swizzling (also called interleaving or tiling) to allow reading or writing a 2D footprint of pixels in a single access [McCormack et al. 1998]. Lossy compression has been used for textures since early GPU designs to reduce memory bandwidth and, therefore, power [Iourcha et al. 1999], while lossless compression is increasingly being used to extend the benefits of compression from textures to other kinds of data. This can reduce access bandwidth by up to 50% [ARM 2017].

I3D '20, May 5–7, 2020, San Francisco, CA, USA

ACM ISBN 978-1-4503-7589-4/20/05.
https://doi.org/10.1145/3384382.3384533

Recent GPUs also support shared virtual memory [Intel 2016]. This allows the CPU and GPU to use the same virtual address pointers so that memory resources may be shared between the CPU and GPU. Reducing the cost of coherency between CPU and GPU caches is an ongoing research topic [Power et al. 2013].

Since shared virtual memory allows application software to directly access GPU data in memory, graphics drivers cannot hide the use of swizzling or compression. Current APIs expose the swizzle format to application software [Grajewski 2012; Microsoft 2018]. Deswizzling in software makes the code more complex and makes it hardware dependent unless the swizzle pattern is widely adopted. Decompressing in software is even more complex. This problem also applies inside the GPU, since some GPU blocks need to access compressed data directly to reduce bandwidth on internal buses. Typically, this requires that all blocks in the GPU support accessing compressed data for compression to be used [Brennan 2016].

We propose modifications to the virtual address translation components of modern CPUs, defining a new page translation system that enables decompression/deswizzling on data access. Our system

(1) Allows CPU software to access block-compressed & swizzled GPU data as if it is stored uncompressed & unswizzled,
(2) Fully supports cache coherent shared virtual memory between multiple integrated CPUs and GPUs,
(3) Allows resource-dependent compression/swizzling to be selected for an unbounded number of memory resources,
(4) Increases CPU software rendering efficiency compared to accessing uncompressed/unswizzled data,
(5) Allows compressed resource management to be performed in app/driver code instead of kernel code, and
(6) Adds no memory latency for accessing regular (i.e. uncompressed/unswizzled) data.

2 BACKGROUND

Before we discuss the details of our approach, in this section we provide an overview of current practices on compression and swizzling in existing GPUs and methods for providing CPU access to GPU resources when CPUs and GPUs share the same memory. We also provide an overview of the virtual address translation process in current CPUs, as this paper proposes modifications to this process.

2.1 Integrating CPUs and GPUs

Typically CPU/GPU shared virtual memory is implemented by integrating CPUs and GPUs onto the same die and allowing them to access a common memory. CPUs and GPUs can be integrated onto the same die in several ways. Figure 1 illustrates several alternatives. On the left we have a CPU and a GPU sharing access to on-chip SRAM. Each of them includes an L1 cache and multi-level page translation logic, referred to as a *page walker*. On the right we have multiple CPU cores and independent GPU blocks that share access

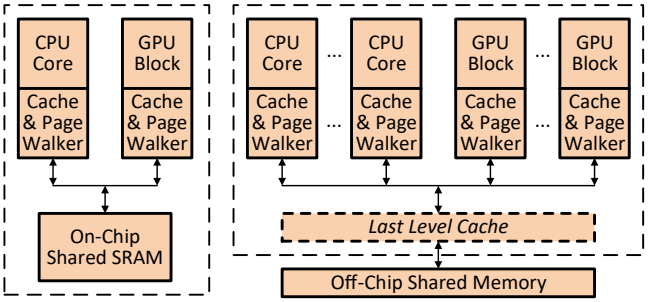


Figure 1: *Two examples of how to integrate CPUs and GPUs onto a single die for shared virtual memory. The shared memory may be on-chip or off-chip, optionally with a last-level cache for higher performance configurations. Each CPU core or independent GPU block typically has its own independent caches and page translation logic (*page walker*), but page translation must be centrally managed.*

to off-chip memory. Each CPU core typically has both L1 and L2 caches as well as its own page walker logic. Similarly, the GPUs may be organized into multiple blocks with independent L1/L2 caches and their own page walker logic.

Although these configurations contain multiple page walkers, the paging system in a shared virtual memory system must be centrally managed, typically by the operating system. This allows the CPUs and GPUs to use the same virtual address to access the same memory location. This is different from integrated configurations that divide up the memory into a portion owned by the CPUs and a portion owned by the GPUs. In such systems data typically must be copied to be converted between CPU and GPU access. Shared virtual memory avoids the copy, but data must still be decompressed for CPU access and CPUs must be able to access the data in swizzled order. Attempts to define standard swizzle patterns have met limited success [Microsoft 2018]. Compression standards also exist, but GPU vendors do not restrict themselves to these standards, especially for lossless compression.

This paper solves these problems by allowing CPUs to access GPU data that is stored in compressed and swizzled forms. No standardization is required since the CPUs only need to support the compression and swizzling methods used by the GPUs that are integrated into the same die.

2.2 Types of Compression in GPUs

Modern GPUs support a variety of both lossy and lossless compression methods. Lossy compression typically reduces the data footprint by a fixed amount, at the cost of approximating the data. For that reason, lossy compression is usually specified in the graphics API so that users are in control of the degree of approximation and the kinds of visual artifacts that may be introduced.

Lossless compression, on the other hand, exactly reproduces the original data. As a result, GPUs can use lossless compression without exposing compression in the graphics API. In general, modern GPUs use lossless compression whenever the bandwidth reduction is of great enough benefit and where the added latency of compression and decompression can be hidden. Since GPUs are designed to hide latency, lossless compression can be used for almost all data.

Another important criteria for compression is whether the data must be decompressed sequentially or whether random access is supported. Video compression methods, for example, are designed to be decompressed sequentially. As a result, the compression method can use data from earlier in the frame or from previous frames. But data structures such as textures, frame buffers, and index buffers are generally not accessed sequentially, but must be able to be accessed at arbitrary locations. To permit this, the data to be compressed is divided into blocks, each of which is compressed independently of the others, so that only the block containing the desired data needs to be compressed or decompressed on access.

GPU shader instruction sets provide many ways to indirectly access memory by using a memory resource descriptor with an array index. The descriptor specifies the array size, pixel format, type of swizzling, compression type, and base addresses for data and meta-data, if any. Dedicated logic uses the descriptor to convert array indices into memory addresses for both the compression data and the meta-data and performs the necessary compression and decompression. This allows compression to be used with 1D buffers, as well as 2D and 3D pixel arrays.

2.3 Block Compression in GPUs

GPUs use both lossless and lossy block compression for reducing memory bandwidth. Each data block is compressed independently to allow random access. A typical example is AMD's Delta Color Compression (DCC) [Brennan 2016].

When using lossless block compression, not all data blocks can be compressed the same amount. Some data blocks cannot be compressed at all, so they must be stored in uncompressed form. As a consequence, information about how a block is compressed (or not) cannot fit into the original data footprint.

The typical solution to this problem is to add a separate array of per-block meta-data bits that specify the type of compression used in each compression block. Then compressed data is stored in an array the size of the uncompressed data. An example lossless block compression scheme is shown in Figure 2. In 4:4 mode, only the memory units that are accessed need to be read or written. In 4:2 and 4:1 modes, all of the compressed data in a block must be read or written to access any part of the block. 4:0 mode selects a constant value for the entire block. This can be a fixed clear color. In DCC the clear color can be specified in the descriptor, though compression is not supported for all memory access agents [Brennan 2016].

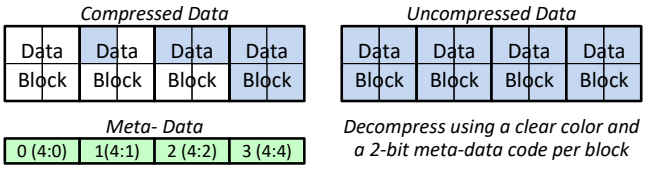


Figure 2: *An example of lossless block compression. Each block is 256B and consists of four 64B memory accesses, storing* 8×8 *32-bit pixels. The level of compression is specified by two meta-data bits per block, indicating possible compression ratios 4:0, 4:1, 4:2, and 4:4. Note that 4:0 uses a default clear color for the entire block (e.g. black).*

For lossy block compression, each block is typically compressed by the same amount to simplify random access. Common GPU lossy

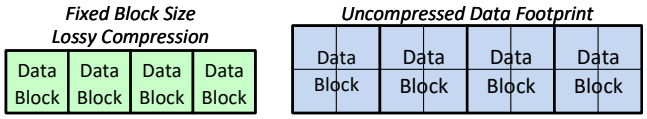


Figure 3: *An example of lossy block compression. Each compressed block is 64B and decompresses to 256B that represents* 8×8 *32-bit pixels. Accessing the uncompressed data using memory addresses requires a second address space the size of the uncompressed data.*

compression methods use a 4:1 compression factor, as illustrated in Figure 3. Two address spaces are required: one for the lossy compressed data and a second the size of the uncompressed data.

Typically, lossy compression is used for read-only data and only when the application chooses to use lossy compression. This is due to the visual artifacts that can result on compression. The most common usage is for pre-compressed texture data. For this reason, we did not evaluate performance for accessing lossy compressed data from CPU software.

Finally, the multiple pixel formats supported by GPUs may be considered to be a form of lossy compression. For example, R4G4B4A4 is a 2:1 lossy compression of R8G8B8A8. Supporting multiple pixel formats in software is not as big a problem as supporting other compression formats for three reasons: (1) they are standardized, (2) the software system usually controls which pixel format is used, and (3) only a few are used regularly. Still, it is simple to define lossy compression formats that convert smaller formats to 8-bit/16-bit integer or 16-bit/32-bit floating point components on CPU reads and convert back to smaller formats on CPU writes, possibly in addition to other forms of lossy compression.

2.4 Address Swizzling in GPUs

GPUs have long used address swizzling [Giesen 2011] to allow reading or writing a 2D footprint of pixels in one memory access. Address swizzling converts an (x,y) or (x,y,z) array index into a memory address by interleaving low order index bits into low order bits of the array address. The result is that each memory access covers a 2D or 3D footprint of the array.

For example, Microsoft Direct3D 12 [Microsoft 2018] defines a set of standard swizzle patterns that pack 4×4 32-bit pixels into 64B, 8×8 32-bit pixels into 256B, etc. Figure 4 illustrates this swizzle pattern in terms of arrays of pixels (left) and sequences of bytes (right). Note that it does not matter for bandwidth how pixels are packed within a single memory access.

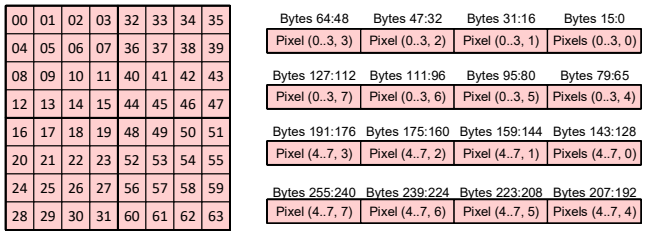


Figure 4: *The Direct3D 12 swizzle pattern for an 8x8 array of 32-bit pixels. The left side shows an 8x8 array of pixels with the word offset for each pixel in the 8x8 array. Dark boxes outline 64B memory access units. The right side shows the mapping of pixels to 256B of memory. Each row is a sequential 64B memory access unit.*

Each address swizzling pattern defines a maximum *swizzle tile* size within which index bit interleaving occurs. For Direct3D 12, the size is 64KB, which interleaves 128x128 32-bit pixels. Above that level, data is stored as a linear array of swizzle tiles. Arrays must be padded so that the array dimensions are multiples of the width and height of a single swizzle tile, which is usually also the memory allocation unit size.

2.5 CPU access to GPU resources

At present there is no good way to provide CPU access to compressed and/or swizzled GPU resources. A common technique is to decompress and deswizzle while copying GPU data to CPU memory, and vice versa when copying the data back. Copying forces exclusive access by either the CPU or GPU, in addition to the latency and power cost of copying the data.

Another common technique is to define a limited number of special *memory apertures* on the CPU that each have an associated resource descriptor. Memory accesses within the aperture's address range use the descriptor to convert addresses to (x,y) array indices, which are converted to addresses in swizzled memory. This technique has high latency and does not permit cache coherency.

Universal compression has been proposed [Verrilli et al. 2016], which stores extra meta-data bits for each block of physical memory. Compression and decompression are then handled in the CPU memory controller. This requires using a common compression method for all of memory, instead of allowing different methods for different memory resources. It also stores compression meta-data for all of physical memory, which increases the memory cost in regions of memory where compression is not used.

Another proposal is to use the existing page table entries to directly store meta-data bits, e.g. by stealing unused high order physical address bits [Malyugin et al. 2019] or by expanding the page table to include all of the per-block meta-data [Ekman and Stenstrom 2005]. The limited number of bits available per page restrict the former to a small number of choices for compressing an entire page. The latter dramatically increases the complexity of page translation, especially for multiple CPU systems that need to synchronize independent page walks. It also does not support GPU access to an array of meta-data. Both methods require the kernel-mode paging system to manage compression.

2.6 Virtual Address Translation

Our solution for allowing CPU access to GPU resources involves modifying the virtual address translation operations of CPUs. The page translation systems used on modern CPUs generally implement a 48-bit virtual address that is mapped to 4KB pages of physical memory. As a result, 4KB is the largest unit of memory that can be sequentially physically accessed. The translation uses a sequence of page table look-ups, where each page table is necessarily limited to 4KB. This sequence of look-ups is referred to as a *page walk*.

Figure 5 shows the essential features of each page table entry. The high order *Page Frame* bits specify a 4KB aligned physical address. The total number of Page Frame bits supported depends on how large a physical address space the CPU supports. The low

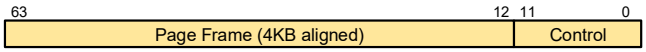


Figure 5: *Abstracted form of a typical 64-bit page table entry in current CPUs. Control bits specify page protections and how to perform the page walk. Page Frame bits specify a 4KB-aligned page address.*

order *Control* bits determine how the page translation is performed and specify various kinds of protections and state. Typical Control bits include write-enable, execute-enable and user mode, which specifies whether the page can be accessed by user code or only kernel code. Bits are also present to record whether data referenced by the page has been read or written.

Figure 6 illustrates using four levels of tree-structured page table lookups to map a 48-bit virtual address into a physical address. Each level uses 9-bits of the virtual address to select a table entry from a specified 4KB page. The PML4, PDPTE and PDE levels (the names are those used in the IA-64 paging system) each read a table entry that specifies the Page Frame for another page table. The PTE level reads a table entry that specifies the Page Frame of the mapped physical address.

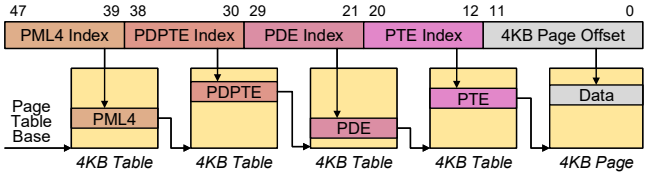


Figure 6: *Four-level page walk used in current Intel/AMD CPUs to map a 48-bit virtual address into a physical address. Physical memory is allocated in units of aligned 4KB pages.*

Addressing virtual data, therefore, requires four page table lookups to determine the physical address. CPUs minimize this cost by caching the individual page table lookups. More significantly, CPUs use a *translation lookaside buffer* (TLB) to allow matching the upper 36-bits of a 48-bit virtual address to a physical address. Any page in the TLB can be accessed without doing any table lookups.

3 COMPRESSION WITH PAGE TRANSLATION

Using memory resource descriptors is not an option on CPUs. Application code running on CPUs directly computes memory addresses from internal array indices, without any way to reference to a memory resource descriptor.

As an alternative to a memory resource descriptor, we propose to take advantage of the CPU's virtual to physical page translation system. Figure 7 shows a page translation that maps each page address in an uncompressed address space to a full page (using a blue arrow) and a partial page (using a green arrow).

For lossless compression (on the left), the blue arrows point to pages of compression blocks and the green arrows point to the subset of a page of meta-data needed to interpret each compressed page. Each 256B compression block requires 2 meta-data bits, so a 4KB page of compressed data requires 32-bits of meta-data. A 64KB page would require 512 meta-data bits.

For lossy compression (on the right), the green arrows each point a quarter of a page of compressed data. The blue arrows point to

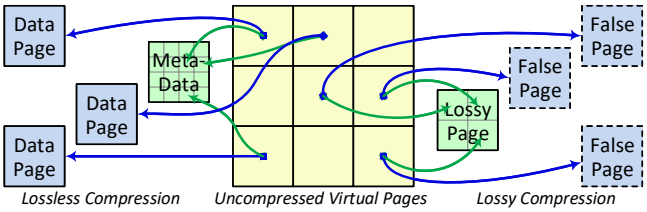


Figure 7: *(Center) virtual pages of uncompressed data are (left) mapped to both a page of lossless block compressed data and a subset of a page of meta-data and are (right) mapped to a subset of a page of lossy compressed data and a false page that provides an address range for the uncompressed data that is not mapped to memory.*

false pages that are not mapped to memory – these pages provide an address range to reference the uncompressed data.

Thus, we propose using a scheme that allows two page pointers per page table entry. This allows CPU software to compute a memory address that the page translation system can use to access both arrays, so that dedicated hardware can automatically perform the compression and decompression, as happens in GPUs.

3.1 New 64-bit Page Translation System

A page translation system that supports automatic compression and decompression on a CPU must be at least as good as current page translation systems when accessing ordinary data. More specifically, it must meet the following requirements:

(1) The new system must not increase the latency to access uncompressed data
(2) The new system must not increase the bandwidth required to support page translation
(3) The new system must not increase the delay through logic in the paging system
(4) The new system must be fully backwards compatible with existing application software
(5) The new system must be fully cache coherent for all existing CPU configurations
(6) The new system must provide specific benefits that justify changing CPU paging

We propose a new page translation system that uses 64KB pages to map a full 64-bit virtual address space. 64KB pages have significant benefits compared to 4KB pages. They dramatically increase the efficiency of the TLB, since each entry covers 16 times as much data. They also greatly reduce the overhead of page management. Virtual address space has long been allocated in 64KB chunks in Windows and can be in Linux, so the larger page size does not waste address space. The only issue is compatibility with 4KB page management, which is covered in Section 5.3.

In this new system, each 64KB page stores 4096 128-bit page table entries instead of using a 4KB page to store 512 64-bit entries. These larger tables make page translation significantly more efficient despite the 128-bit table entries, since 12-bits can be translated per table lookup, instead of 9-bits. Figure 8 illustrates using four levels of tree-structured page table lookups to map a 64-bit virtual address into a physical address. This supports enterprise level applications that require more than the current 48-bit address range. Normal CPUs can use the new system with just three lookup levels for

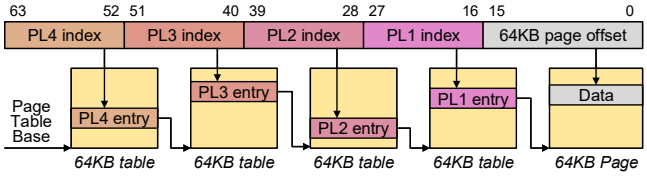


Figure 8: *Four-level page walk used to map a 64-bit virtual address into a physical address. Physical memory is allocated in units of aligned 64KB pages. Fewer levels may be used when 52-bit, 40-bit, or 28-bit virtual address ranges are sufficient.*

52-bit virtual addresses, or with even fewer levels for SOCs and embedded processors.

Figure 9 shows the essential features of each 128-bit page table entry. The Page Frame now specifies a 64KB aligned address instead of 4KB aligned. This allows four additional Control bits, which can be used to define the kind of address swizzling used, as described in Section 5. The Subpage Frame specifies a 16B aligned address. At level PL1, this can be used to reference the meta-data associated with a 64KB compressed data page. Finally, the Select (Sel) field specifies what type of compression the data uses, where zero means that there is no compression and other values select up to 15 different compression methods. This should be sufficient to specify the range of compression modes provided by the GPUs that are embedded with the CPU and therefore share access to the same physical memory.

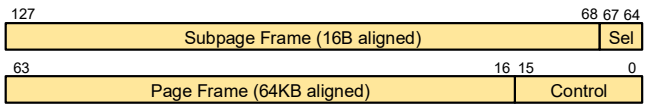


Figure 9: *Example 128-bit table entry format for 64KB pages, including a reference to a 64KB page and to a 16B aligned subset of a second 64KB page. These may be used to reference a page of lossless compressed data blocks and a sub-page of the corresponding meta-data.*

3.2 Multi-Stage Address Translation

Modern paging systems on both CPUs and GPUs sometimes support multi-stage address translation. An example is when the paging hardware directly supports a hypervisor. First a guest address space is converted to a host address using a normal multi-level lookup. This serves as a virtual to physical translation for the hypervisor. But then a second multi-level lookup is performed to convert the host address into a physical address. This serves as the virtual to physical translation for the host operating system. The result is that a hypervisor operating system runs on top of a host operating system, with full protection of physical memory from disallowed accesses by the hypervisor.

Many GPUs now support a multi-stage address translation feature called *tiled resources* [Microsoft 2013] or *sparse textures* [Sellers 2012], which was previously implemented in software under the name Mega-Textures [Connery 2007]. Briefly, this divides a graphics resource into 64KB tiles so that the application can select which tiles it needs for a given frame. The application uses a tile translation table to convert address ranges for those tiles into virtual addresses, which the operating system converts to physical memory addresses using its own page tables. This allows the graphics driver and applications to perform their own page translation, while leaving physical address translation and physical page protection to the operating system.

Multi-stage address translation can be used with this new system in the same way that it is used to separate hypervisor address translation from host operating system translation in existing CPUs. This method can also be used to implement tiled resources/sparse textures on CPUs. Similarly, the stage that maps an uncompressed address range to compressed data and meta-data can be separate from the stage that maps virtual addresses to physical addresses. This allows block compressed data management to occur in user-mode graphics drivers and applications, while permitting the operating system to retain full control of the physical address space.

4 IMPLEMENTING BLOCK COMPRESSION

This section describes a basic structure for implementing block compression algorithms in the CPU by means of information stored in the page table.

4.1 Compression/Decompression Logic

Supporting automatic block compression requires only localized changes to the processor's memory datapath. Solid boxes in Figure 10 show the basic data flow for a processor (CPU or GPU) that communicates with shared memory through local L1 and L2 caches. The dotted box shows where the new logic interacts with that data flow. Actual GPUs and CPUs are more complex [Junkins 2015].

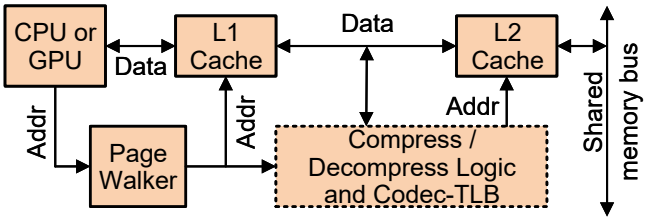


Figure 10: *Lossless compression and decompression occur between the L1 and L2 caches, when page translation (performed by the page walker block) determines that the data is stored compressed in memory. Dotted lines mark the block that is added to the hardware. The datapath and latency for uncompressed data is unchanged.*

Figure 11 shows a more detailed view of how the caches and compression logic interact with each other. First, consider a read of compressed data that misses in the L1 cache. In that case, the following sequence of steps occur:

(1) The page walk identifies the page as storing compressed data and loads the the *Codec TLB* with the necessary compression information into the *Codec TLB*, if it is not already there.
(2) Next, the compression block's meta-data is read from the *Meta L1 Cache*, loading it from the L2 cache if necessary. The location of the meta-data is specified in the Codec TLB entry.
(3) The meta-data determines which compression data to read from the L2 cache, loading it from memory if necessary.
(4) The Compressor/Decompressor block decompresses the data and stores the block into the L1 cache, along with tags that specify the Codec TLB entry for each L1 cacheline.

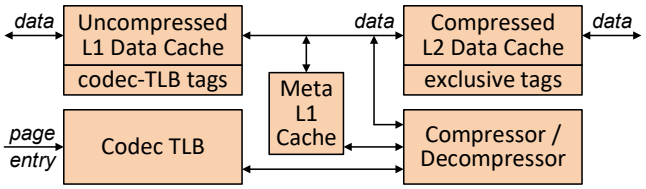


Figure 11: *Details of compression/decompression logic and how it interacts with the L1 and L2 caches. The Codec-TLB stores compression information for active blocks. The Meta L1 cache stores meta-data. The exclusive tags on the L2 cache are used for cache coherency.*

Writes to the L1 cache do not cause any action other than loading an entry into the Code TLB, if necessary, so that the L1 cacheline can be tagged with its compression data. Our method assumes that the L1 is a write-back cache, which is common in modern CPUs. As a result, recompression does not occur until cachelines are evicted from the L1 cache. At that point the following sequence of events occur:

(1) Any L1 cachelines in the compression block are evicted and are sent to the Compressor/Decompressor.
(2) If any part of the block is not in the L1 cache (e.g., that cacheline was not written), the block is decompressed from the L2 cache to fill in the missing data.
(3) The Compressor/Decompressor compresses the block and writes out compression data and meta-data to their caches.

Optimizations and simplifications are possible for the read and write paths. For example, if the meta-data is not available in the Meta L1 or L2 caches on a read, both the meta-data and the complete compression block can be read from memory in parallel. This results in reading unnecessary compression block data if the block is compressed below 4:4, but it avoids doubling the memory latency by reading the meta-data before reading the compression data.

Finally, when a Codec TLB entry is evicted, all L1 data that references that entry must be evicted, along with the corresponding main TLB entry. This forces any future access to that page to do a page walk that reloads compression data into the Codec TLB.

4.2 Cache Coherency for Compression

Hardware support for cache coherency is important, since the alternative is for software to enforce coherency. Some kind of synchronization is required when multiple processors access the same data words, but software cache coherency requires cache flushes to ensure that dirty data does not remain in one of the caches.

Cache coherency is complicated by several factors. CPU/GPU systems may contain multiple L1 caches. They may even contain multiple L2 caches, e.g. Intel® multi-core CPUs that provide a separate L2 cache per processor on a ring bus. However, existing shared cache exclusive access protocols suffice for enforcing cache coherency for data that is not compressed.

When data is stored compressed, writing any part of a compression block requires exclusive access to the entire block and its meta-data. Our method expands the unit of exclusive access from a single memory access unit (e.g. 64B) to an entire compression block. The size is known from the Codec TLB entry for that block. As a result, writing to any byte within a compression block uses the exclusive access protocol to flush the entire compression block from any other processor that has that block in its caches.

It is also necessary to gain exclusive access to the meta-data bits associated with the block. This is complicated by the fact that a single cacheline of meta-data bits can cover a lot of compression blocks, e.g. 256 compression blocks for 2-bit meta-data and a 64B cacheline. Rather than requiring exclusive access to such a large region of memory, we propose marking cachelines of meta-data bits in the L2 cache. An exclusive access request to a meta-data cacheline would refer to the L1 Meta Cache, which can store exclusive access bits for individual compression blocks or for small sets of compression blocks. Note that invalidating meta-data bits due to exclusive access requires flushing any compression data associated with those bits.

5 AUTOMATIC ADDRESS SWIZZLING

Automatically compressing and decompressing data is not sufficient, since GPUs typically swizzle x and y index bits to store 2D data. Figure 4 illustrates a swizzle pattern for an 8x8 block of 32-bit pixels.

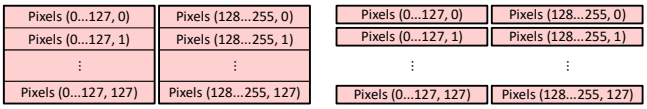
We describe two ways to automatically swizzle and deswizzle data on access by the CPU. The first requires changing application software to be aware of the 64KB page size and the second does not. We then describe an extension to our page translation scheme that supports the second method.

5.1 Within-Page Deswizzling

Accessing a swizzled array from a CPU is problematic. Application code can deswizzle the data, but that can require extra inner-loop instructions. Worse, it may make the code hardware-dependent because swizzle patterns are not well standardized across GPU vendors and may change from generation to generation.

A simple change to the Compression/Decompression logic in Figure 11 allows accessing data in row-major order within each 64KB page. The swizzle pattern is selected in the Control bits of the PL1 page table entry. Then the page offset bits are reordered to map the swizzled data in the page into a row-major order array. For 32-bit pixels, this produces a 128×128 pixel array per page. Software then uses high order X and Y address bits to select a page and uses the lower seven of the x and y address bits to perform ordinary row-major order pixel accesses within the page.

Figure 12a illustrates 32-bit pixels in 64KB pages in deswizzled order. For within-page deswizzling, the application software does not need to know anything about the actual swizzle pattern. However, it needs to know how large an array of pixels maps to a page and needs to organize its loops around that per-page array size. The following section describes how to convert the entire array to row-major order, rather than just deswizzling within 64KB pages.



(a) Two deswizzled 64KB pages (b) 256 separate 512B pages

Figure 12: *Pixels in pages: (a) two 64KB pages of 32-bit pixels that have been within-page deswizzled to 128x128 row-major order arrays, and (b) dividing those two 64KB pages into 256 512B pages, each of which stores 128 32-bit pixels from a single row.*

5.2 Across-Page Deswizzling

Within-page deswizzling requires writing software applications with explicit knowledge of the 64KB page size. While this may not be very difficult, it does not provide any particular advantage to do so, except the advantage of not needing to write the code to support all possible swizzle patterns that it might be used with. There are two other ways to write CPU software to access 2D arrays of data:

(1) Row-major order: for scanline-based algorithms, it is simple and efficient to access arrays without any swizzling. All programming languages support row-major order by default.
(2) 2D-block order: for rendering algorithms, memory bandwidth can be reduced by swizzling data within small blocks, e.g. one to four cachelines. All GPUs support swizzling because it is more efficient for rendering.

First, consider row-major order (1). Figure 12b illustrates accessing 32-bit pixels from the within-page deswizzled data illustrated in Figure 12a. After every 512B (128 pixels), stepping through the pixels in row order requires changing pages, because pixel (127,0) and pixel (128,0) are in different 64KB pages. Therefore, we can create row-major pixel order in a virtual address space if we use 512B pages. Note that we do not need to allocate memory on 512B boundaries to achieve this: the data is still stored in 64KB pages. We just need to be able to perform page translation at 512B boundaries.

Now consider 2D-block order (2). With 32-bit pixels, 4x4 pixels fit into a 64B cacheline and 8x8 pixels fit into four cachelines. Storing a 4x4 block per 64B cacheline has numerous advantages for rendering algorithms, as compared to storing 16x1 pixels in a 64B cacheline. For example, suppose we are rendering a 7x7 pixel region. That touches exactly four aligned 4x4 pixel blocks. It touches seven to fourteen 16x1 pixel blocks. On average, storing the data as 16x1 pixels per cacheline instead of 4x4 pixels per cacheline requires accessing 2.4 times as many cachelines.

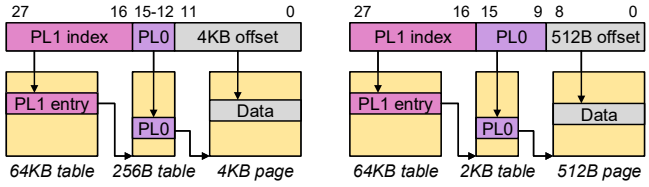
Both of these usages require a way to support page translation at a finer granularity than 64KB. The following section describes a way to achieve this without requiring memory allocation at a finer granularity than 64KB.

5.3 Variable Size PL0 Page Walk

Section 3.1 refers to needing to support 4KB page translations for backward compatibility with existing paging systems. The solution in the previous section also requires support for pages smaller than 64KB in order to support across-page deswizzling, e.g. 512B page translations. But it would be very costly to require the operating system to support allocating memory in multiple page sizes.

The solution to these problems is to allow dividing 64KB pages into smaller *subpages*, each of which can specify its own aligned addresses and page protections. A subpage is not an allocation unit: memory is allocated only in 64KB chunks. Instead, subpages are a way to map subsets of a 64KB page to different ranges of the virtual address space.

Figure 13 illustrates two examples of how to modify the PL1 page walk shown in Figure 8 to allow an extra PL0 subpage lookup after the PL1 lookup. Control bits determine whether the PL1 lookup produces a data page or accesses a table for a PL0 page walk and how many bits to map. The PL0 index can contain from 1-bit to 16-bits. This produces subpage sizes from 32KB down to 16B, using table sizes from 32B to 64KB (2 to 4K table entries).



(a) 4-bit PL0 index (4KB subpage) (b) 7-bit PL0 index (512B subpage)

Figure 13: *A level zero lookup allows the tile translation table to map subpages that are subsets of 64KB memory allocation units. The number of PL0 index bits used determines the subpage size: (a) 4-bit PL0 index that divides a 64KB page into 16 4KB subpages, and (b) 7-bit PL0 index that divides a 64KB page into 128 512B subpages.*

Figure 13a shows a PL0 access that allows full compatibility with hardware or application software that uses 4KB pages. Each table entry allows a 4KB-aligned sub-page reference with a complete set of page protection bits. This can be used to map page control bits and an arbitrary 4KB-aligned address per page, so long as the operating system allocates virtual memory in 64KB chunks, as Windows does [Chen 2003] and as Linux can do.

Figure 13b shows a PL0 access that allows row-major deswizzling for 32-bit pixels, as shown in Figure 12(b). Other pixel sizes or swizzle patterns may require other subpage sizes, as does 2D-block order partial deswizzling. For example, 4x4 2D-block order requires a 2KB subpage size, which would use a 5-bit PL0 index. 8x8 2D-block order requires a 4KB subpage size, as illustrated in Figure 13(a).

6 EVALUATION

A key goal of this work is to enable applications to perform a mix of processing on GPUs and CPUs, depending on which processor is more power efficient for each task. For that to be practical it must be possible for each processor to access the data without performance loss or power gain due to decompressing data before CPU use or due to refraining from using compression on the GPU.

Therefore, for our evaluation we wanted to mock up an application that renders a pixel array on a GPU and then transfers the data to the CPU for further rendering. In that way the data arrives at the CPU already compressed and swizzled. We chose text rendering for the CPU task, since at one time text rendering was often done on the CPU. We compare the following cases:

(1) *Baseline*: CPU accesses row-major uncompressed data that is assumed to have been copied after rendering on a GPU.
(2) *Our method*: CPU accesses row-major order decompressed data that is assumed to have been rendered on a GPU.
(3) *Our method*: CPU accesses 2D-block order decompressed data that is assumed to have been rendered on a GPU.

We also measured the cost of copying the GPU data in order to unswizzle and uncompress it for the baseline case. The results show that our method (2) uses about 5/6 the power and computation time of baseline (1) for row-major pixel order and about 2/3 the power and computation time of baseline (1) when the code is modified to use 2D-block order (3), so that each cacheline access returns a 2D array of texels instead of a linear array of texels. These numbers

are without even considering the cost of copying the data to decompress and partially or fully deswizzle it. The following subsections introduce the test case, test dataset, hardware configuration, and test procedure before the results are discussed.

6.1 Text Rendering Test Case

To evaluate the performance of our approach, we use a test application where the CPU rasterizes random text on a set of textures (Figure 14). CPU rasterization of text is a useful application, since it parallelizes poorly on the GPU. The textures are stored in the GPU memory in a shared virtual memory system. During text rasterization the shared memory containing the underlying textures is read and written on the CPU side. Read is required since the text contains an alpha channel for anti-aliasing, which requires blending with the target texture.

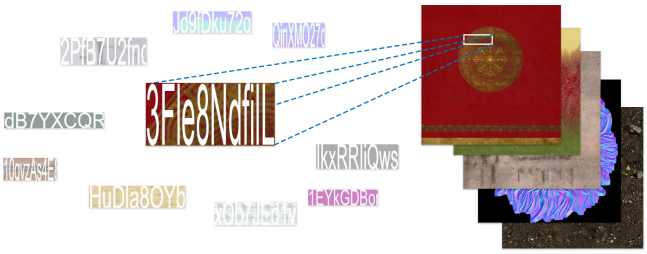


Figure 14: *Our test rasterizes anti-aliased random alphanumeric text on a set of 668 textures, including albedo maps, albedo maps, specular maps, normal maps, and opacity maps. At each iteration an image file is randomly selected and a text box of random size between 64×16 and 256×64 is generated at a random position in the image.*

The dataset used in our tests consists of 668 textures coming from 4 different 3D scenes (Amazon Lumberyard Bistro, Crytek Sponza, San Miguel [McGuire 2017], and Zero Day [Beeple 2015]) that contains albedo maps, specular maps, normal maps, and opacity maps, with decompressed sizes summing to over 5GB. All pixels are stored as 32-bit RGBA values.

Since the textures contain a variety of compression characteristics, rendering text on them simulates the case of web page co-rendering, an important application of our method. In web page font rendering where CPU rasterization can provide more flexibility, our method allows the CPU to directly rasterize fonts on the compressed and swizzled images in shared virtual memory, thus avoiding expensive data copying.

We compare the performance of our method to a *baseline* that runs CPU code for text rasterization, using row-major order to read and write the texture data. The difference between our method and the baseline is the way that the texture data is stored. For our method, we assume that the textures are stored in our swizzled and lossless block compressed format during CPU text rasterization.

For the baseline, we assume that textures are uncompressed and stored linearly in the shared memory, so it does not include our modifications to virtual address translation, as they are not needed. Thus, our baseline imitates current CPU systems that cannot handle compressed and swizzled GPU resources and, therefore, they would either involve copying data to the CPU memory (during which the data is decompressed and deswizzled) or require the GPU to work with uncompressed and deswizzled data (with negative GPU performance implications).

Table 1: *Detailed configuration of different hardware components.*

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

6.2 Hardware Configuration

The simulated CPU is a simplified version of the standard 2010 Intel core i7 configuration, which includes a pipelined L1 cache and an L2 cache, but no L3 cache. The detailed configuration can be found in Table 1. The simulated CPU has a clock speed of 3.2 GHz and DRAM is a single channel DDR3-1600 with 16GB capacity. The CPU pipeline is simplified such that it stalls for every load instruction until the result comes back from lower caches or the main memory (simulated using USIMM [Chatterjee et al. 2012]).

The simulated CPU has a two-level TLB which caches the results of page translations. For the baseline (1), the page size is 4KB (1K pixels). For our method, the page size is 64KB, with a subpage size of 512B (128 pixels) for row-major (2) and a subpage size of 4KB (16 blocks of 8x8 pixels) for 2D-block order (3). Additionally, we add the proposed Meta L1 cache and hardware codec to the chip. We do not model the Codec TLB, instead assuming that the corresponding metadata location of an L1 cache entry is always available. This should have a minor effect since the overhead of reading the codec-TLB is small compared to compression and decompression.

Commercial GPUs use proprietary lossless compression schemes, so we needed to create our own. We use Huffman coding with a fixed Huffman tree for the fixed function lossless compression logic in our simulated hardware. We chose Huffman coding due to its simplicity, which minimizes the codec overhead while achieving a roughly 2:1 average compression ratio on a block size of 8×8 32-bit pixels or 256B. This compression method, block size, and compression ratio are similar to many GPU lossless block compression algorithms.

We estimate the latency of the codec in our hardware to be 50 cycles to decompress a block. This is based on an implementation of static Huffman encoding hardware using barrel shifters [Lee and Park 2004] and a hardwired Paeth codec [Hakkennes and Vassiliadis 1999]. Codec energy consumption is estimated as 1.9947 nJ by using Cacti 7 [Balasubramonian et al. 2017] and summing the energy of all hardware components. Multiple codecs could be used to increase bandwidth between the L1 and L2 caches, if needed.

6.3 Test Procedure and Results

Before text rasterization, the dataset has an average block compression ratio of 2.39. Some textures (e.g. albedo maps) often contain high frequency details that result in a low lossless compression rate, while other textures (e.g. normal maps) usually contain large regions with constant data, leading to a high compression rate.

Our test application randomly selects an image file and generates a text box at a random position in the image with a random size

Table 2: *Energy Breakdown of hardware components in millijoules, with ratios of our methods to the baseline.*

	(1) Baseline 4KB pages		**(2) Our Method** 64KB pages, 512B subpages			**(3) Our Method** 64KB pages, 4KB subpages		
	energy	*%*	*energy*	*%*	*ratio*	*energy*	*%*	*ratio*
TLB	0.01	0.00%	0.03	0.01%	2.15	0.00	0.00%	0.31
L2-TLB	0.02	0.01%	0.04	0.02%	2.05	0.01	0.00%	0.30
Data L1	10.67	3.69%	10.78	4.49%	1.01	10.59	5.44%	0.99
Meta L1	0.00	0.00%	0.25	0.10%	∞	0.20	0.10%	∞
L2	1.72	0.59%	1.51	0.63%	0.88	1.16	0.60%	0.68
Codec	0.00	0.00%	6.08	2.53%	∞	4.95	2.54%	∞
Memory	276.29	95.70%	221.27	92.21%	0.80	177.84	91.31%	0.64
Total (rendering)	**288.71**	**100.00%**	**239.96**	**100.00%**	0.83	**194.76**	**100.00%**	0.67
Initialization	**1384.79**	(*extra*)	**0.00**	- - -	0	**0.00**	- - -	0

Table 3: *Time Breakdown in CPU cycles, with ratios of our methods to the baseline.*

	(1) Baseline 4KB pages		**(2) Our Method** 64KB pages, 512B subpages			**(3) Our Method** 64KB pages, 4KB subpages		
	# of cycles	*%*	*# of cycles*	*%*	*ratio*	*# of cycles*	*%*	*ratio*
VA Translation	**38,232K**	**3.59%**	**97,718K**	**10.68%**	2.56	**30,413K**	**4.38%**	0.80
– TLB Read	438K	0.04%	906K	0.10%	2.07	133K	0.02%	0.30
– L2-TLB Read	2,628K	0.25%	5,432K	0.59%	2.07	797K	0.11%	0.30
– Data L1 Read	3,177K	0.30%	7,061K	0.77%	2.22	1,114K	0.16%	0.35
– L2 Read	1,845K	0.17%	5,526K	0.60%	3.00	1,152K	0.17%	0.62
– Memory Read	24,799K	2.33%	59,104K	6.46%	2.38	16,764K	2.41%	0.68
– Queueing Delay	5,344K	0.50%	19,689K	2.15%	3.68	10,453K	1.51%	1.96
Load	**949,099K**	**89.08%**	**747,481K**	**81.69%**	0.79	**595,528K**	**85.75%**	0.63
– Data L1 Read	78,140K	7.33%	71,609K	7.83%	0.92	68,557K	9.87%	0.88
– Meta L1 Read	0	0.00%	7,452K	0.81%	∞	4,965K	0.71%	∞
– L2 Read	44,326K	4.16%	45,099K	4.93%	1.02	29,147K	4.20%	0.66
– Memory Read	721,178K	67.69%	385,532K	42.13%	0.53	320,540K	46.15%	0.44
– Queueing Delay	105,454K	9.90%	144,082K	15.75%	1.37	110,256K	15.88%	1.05
– Codec Delay	0	0.00%	93,152K	10.18%	∞	62,063K	8.94%	∞
Store	**78,140K**	**7.33%**	**70,423K**	**7.70%**	0.90	**68,557K**	**9.87%**	0.88
Total (rendering)	**1,065,470K**	**100.00%**	**915,066K**	**100.00%**	0.86	**694,499K**	**100.00%**	0.65
Initialization	**2,249,921K**	(*extra*)	**0**	- - -	0	**0**	- - -	0

chosen between 64×16 and 256×64. A random string consisting of alphanumeric letters is rasterized with a scale that fills the entire text box using the Arial Black font face, using the FreeType library [FreeType 2018]. The modified image blocks are recompressed when data is flushed from L1, which can cause changes in their compression ratio and meta-data. We repeat this process 10,000 times to collect the statistics as shown in Tables 2 and 3.

When using row-major order (2), our method reduces energy use by 17% and computation time by 14%. This compares running exactly the same row-major text rendering code for baseline and for our method. Virtual address translation is more costly due to the 512B subpage size. This cost could be reduced by coding for within-page deswizzling, which eliminates the 512B subpages.

When using 2D-block order (3), our method reduces energy use by 33% and computation time by 35%, which is a 1.53× speedup. This improvement is due to rewriting the code to access data in 2D-block order, that is, in swizzled 8x8 blocks of data in linear order in memory. This allows using 4KB subpages, so that the page translation cost is now lower than baseline. More significantly, this reduces the CPU cycles for load operations, which dominate the computation time, from 79% of baseline to 63% of baseline. As a result, the total rendering time is reduced by 35% and the total energy use is reduced by 33%, as compared to baseline.

These numbers do not include the overhead of the GPU either using uncompressed/unswizzled data, or else the cost of copying the GPU data in order to convert it to uncompressed and fully or partially unswizzled form before CPU access. The latter cost is shown on the *initialization* lines at the bottom of the two tables. For the baseline, this initialization copy operation requires 2.1× the total cycles for rendering and 4.8× the total dynamic energy for rendering. This is because the memory footprint of copying the whole dataset is much larger than the memory footprint of the font rasterization task. In the 10000 iterations of font rasterization on crops of random images, at most 625 MB of data are read from the memory. In comparison, the initialization process needs to copy the whole dataset, which has 5.7 GB uncompressed size. So our

method of using the page table to decompress and deswizzle data on demand allows most of the data to remain in compressed and swizzled form.

7 DISCUSSION AND FUTURE WORK

Our method opens up new possibilities for interactive rendering with CPU and GPU cooperation. At present opportunities to split work between CPU and GPU are limited by the tremendous cost of restricting CPU data to be uncompressed and unswizzled. Therefore, many tasks have been transferred to the GPU. Our work allows choosing which processor is most efficient for each task, without concern for data transfer or decompression/deswizzling.

Web page font rendering is an example application that can benefit from our proposed hardware modifications. The Chromium browser uses either software rasterization with zero-copy texture transfer or GPU rasterization with precomputed texture atlases for text, which limits the flexibility of font rasterization [Hartmeier 2016]. Our method solves this problem by allowing the CPU and the GPU to rasterize to the same memory resource without copying data and without abandoning compression and swizzling.

There are a number of ways that this work can be extended to increase the variety of compression options and data re-ordering options that can be supported:

- Global as well as per-block meta-data could be encoded, e.g. to specify background colors or Huffman tables.
- Entire pages could be stored compressed and then decompression software could be called on first access.
- Data could be reordered to support both array-of-struct and struct-of-array access.
- The subpage mechanism could be used to support variable rate compression and fine grain access control.

There are also a number of ways the work can be extended to explore additional applications of this method:

- Further analysis of using this method with web page rendering and other rendering algorithms.
- Analysis of supporting sparse textures/tiled resources on CPUs and other multi-stage page mappings.
- Using compression for vertex buffers or other data structures that are consumed by a GPU but generated by a CPU.
- Testing CPU-only algorithms that may benefit from storing data compressed in main memory.
- Transferring data between non-shared memory using compression and leaving it compressed in CPU memory.

8 CONCLUSION

We have described a way to modify the CPU page translation system to support automatic data compression and address swizzling for CPU/GPU shared memory resources. Our proposed method requires a new logic block that processes compressed or swizzled data passing between the L1 and L2 caches, but otherwise has minimal impact on the CPU memory access path.

Our method allows efficient CPU access to swizzled and compressed GPU resources using software that assumes that data is uncompressed and stored either in standard row-major order or is stored in 2D-block order, which is a linear sequence of small swizzled blocks. In our evaluation of CPU text rendering, both CPU cycles and power were reduced to about 5/6 for row-major order and 2/3 for 2D-block order, compared to that required using data that is stored uncompressed and unswizzled. This suggests that many applications can benefit from our method.

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

Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 14.

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

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

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. *USIMM: the utah simulated memory module.* Technical Report. Univ. of Utah.

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

Richard Connery. 2007. MegaTexture in Quake Wars. https://www.beyond3d.com/content/articles/95/1/

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

FreeType. 2018. FreeType Overview. https://www.freetype.org/freetype2/docs/

Fabian Giesen. 2011. Texture tiling and swizzling. https://fgiesen.wordpress.com/2011/01/17/texture-tiling-and-swizzling/

Slawomir Grajewski. 2012. INTEL_map_texture. https://www.khronos.org/registry/OpenGL/extensions/INTEL/INTEL_map_texture.txt

Edwin A Hakkennes and Stamatis Vassiliadis. 1999. Hardwired Paeth codec for portable network graphics (PNG). In *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, Vol. 2. IEEE, 318–325.

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.

Stephen Junkins. 2015. The Compute Architecture of Intel® Processor Graphics Gen9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

Taeyeon Lee and Jaehong Park. 2004. Design and implementation of static Huffman encoding hardware using a parallel shifting algorithm. *IEEE Transactions on Nuclear Science* 51, 5 (2004), 2073–2080.

Vyacheslav Malyugin, Luigi Semenzato, Choon Ping Chng, Santhosh Rao, and Shinye Shiu. 2019. Transparent hardware-assisted memory decompression. US Patent 10,203,901.

Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P Jouppi, and Ken Correll. 1998. Neon: a single-chip 3d workstation graphics accelerator. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware.* ACM, 123–132.

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

Microsoft. 2013. Tiled Resources. https://docs.microsoft.com/en-us/windows/win32/direct3d11/tiled-resources

Microsoft. 2018. UMA Optimizations: CPU Accessible Textures and Standard Swizzle. https://docs.microsoft.com/en-us/windows/win32/direct3d12/default-texture-mapping

Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 457–467.

Graham Sellers. 2012. AMD_sparse_texture. https://www.khronos.org/registry/OpenGL/extensions/AMD/AMD_sparse_texture.txt

Colin Beaton Verrilli, Mattheus Cornelis Antonius Adrianus Heddes, Brian Joel Schuh, Michael Raymond Trombley, and Natarajan Vaidhyanathan. 2016. Providing Memory Bandwidth Compression Using Back-to-Back Read Operation By Compressed Memory Controllers (CMCs) in a Central Processing Unit (CPU)-Based System. US Patent App. 14/844,516.