GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Parallel Texture Compressor

Tim Dettmar

MatrNr: 26327113

Supervisor: Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2023

Abstract

Texture compression is a specialized form of image compression, adapted to the requirements of computer graphics. As opposed to common image compression algorithms, such as JPEG, compressed textures have a deterministic output size, are random-access, and are relatively simple to decode. These properties allow for the fast access required for real-time applications, such as 3D rendering in games and design applications. Of these texture compression formats, ASTC is one of the most complex, allowing it to achieve high image quality at the expense of a severely reduced encoding speed. In order to reduce encoding times, this project implements a parallel ASTC texture compressor with support for OpenMP and MPI parallelization and explores the performance and scalability of this solution.

Contents

List of Tables iii

List of Figures iii

List of Listings iii

List of Abbreviations iv

1 Introduction 1
1.1 Texture Compression . . . 1
1.2 The ASTC format . . . 1
1.2.1 Basics . . . 1
1.2.2 Integer Sequence Encoding . . . 2
1.2.3 Bit Allocation . . . 4
1.2.4 Block Layout . . . 4
1.3 ASTC Compressor . . . 5

2 Serial Implementation 5
2.1 Intermediate Value Conversion . . . 5
2.2 Endpoint Calculation . . . 6
2.3 Partitioning . . . 7
2.4 Quantization . . . 8

3 Parallel Implementation 9
3.1 MPI/OpenMP Hybrid Implementation . . . 9
3.2 Local Worker . . . 11
3.3 MPI Workers . . . 11

4 Performance 12
4.1 Configuration . . . 12
4.2 Threading Performance . . . 13
4.3 MPI Performance . . . 14

5 Conclusion 16

References 17

A Appendix A1
A.1 Acknowledgements . . . A1
A.2 Source Code Repository . . . A1
A.3 MPI Benchmark Commands . . . A1
A.4 ASTC Partition Generation . . . A2
A.5 Precomputing ASTC Partition Mappings . . . A3
A.6 System Topologies . . . A6
A.6.1 Intel System . . . A6
A.6.2 Ampere System . . . A6
A.7 Block ID - Texel Address Conversion . . . A8

List of Tables

1 Efficient BISE Encodings . . . 3

List of Figures

1 Small Firefox icon compressed with S3TC (DXT1) . . . 2
2 Partitioning Example . . . 2
3 Efficiency of various BISE encodings . . . 3
4 Sample 1-partition block layout . . . 4
5 Sample void-extent block layout . . . 5
6 Problematic block encodings . . . 7
7 Sample 4-partition blocks . . . 8
8 Parallel Implementations of mpASTC . . . 9
9 Work/Dispatch Unit Layout . . . 10
10 MPI/Local Worker Buffer Allocation . . . 12
11 Threaded Compressor Performance . . . 13
12 Initial 2-Node Results . . . 14
13 Runtimes for Various Node Counts . . . 15
14 Relative Efficiency for Various Node Counts . . . 15

List of Listings

1 Available quantization levels . . . 8
2 Preset quantization levels . . . 8
3 OpenMP Work Distribution . . . 10
4 Local worker control structure . . . 11
5 MPI Benchmark Commands . . . A1
6 ASTC Partition Generation Code . . . A2
7 ASTC Partition Map Generation . . . A3
8 ASTC Partition Filtering . . . A3
9 Intel System Topology . . . A6
10 Ampere System Topology . . . A6
11 Supplementary block extraction functions . . . A8

List of Abbreviations

HPC High-Performance Computing

BISE Bounded Integer Sequence Encoding

bpt bits per texel

bpp bits per pixel

MSE Mean Squared Error

S3TC S3 Texture Compression

ASTC Adaptive Scalable Texture Compression

PCA Principal Component Analysis

RDMA Remote Direct Memory Access

NUMA Non-Uniform Memory Access

MPI Message-Passing Interface

1 Introduction

1.1 Texture Compression

Fundamentally, texture compression is a form of image compression. However, unlike variable-rate formats such as the JPEG or PNG formats, texture compression algorithms prioritize decoding speed and access latency over pure space efficiency. The most commonly used compressed texture formats, including ASTC, use block-based compression algorithms [Khr, p. 175]. Such algorithms perform compression on independent groups of pixels with a deterministic input:output size ratio. This is well-suited for the highly parallel nature of GPUs, and the fixed size allows for random access. The relative address of a block corresponding to a pixel may be derived from the specification [Khr, pp. 230, 234] as follows: [1]

$$(\lfloor \frac{x}{W_b} \rfloor + \lfloor \frac{y}{H_b} \rfloor \frac{W_t}{W_b}) S_b \tag{1}$$

where:

- x and y are the coordinates of the pixel in row-major order
- W_b and H_b are the width and height of the block in pixels
- W_t is the width of the image
- S_b is the fixed size of the block (ASTC: 16 bytes)

Similarly to image compression, compressed textures are used to reduce the amount of memory and bandwidth required relative to raw data. [Khr, p. 175]. This can, for example, be used to improve the performance of games, or increase their visual fidelity without consuming more resources [Cha].

1.2 The ASTC format

One of the unique features of Adaptive Scalable Texture Compression (ASTC) that, while making it slow to encode, allows it to achieve better compression quality than competing formats, are the variable number of bits available for texel and colour data that may be selected by the compressor [ARMb]. In ASTC terminology, these are known as endpoints and weights [ARMc].

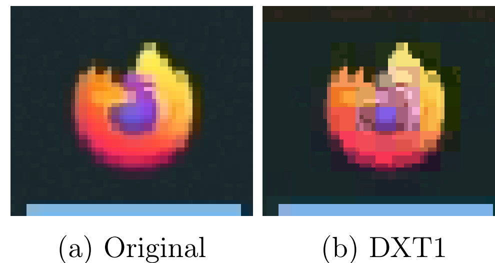
1.2.1 Basics

An endpoint consists of two base colours. To generate more detail, these two colours are then interpolated to generate a gradient of available texel colours. The weight information consists of a sequence of normalized integers describing the texel's colour as the position along this gradient. Weights can be represented with a maximum of 5 bits per texel, and colour representation depends on the encoding mode [ARMc]. In the case of this compressor, the encoding mode chosen allows for up to 8 bits per channel for a maximum

[1] The PVRTC format is one exception: it requires multiple blocks to decode any texel [Khr, p. 263], but this is outside of the scope of the report, and the rule still holds in general.

of 24 bits per colour or 48 bits per endpoint [Khr, p. 241]. ASTC defines many other colour modes [Khr, p. 237], but these modes are out of the scope of this report.

However, blocks in many real-world textures simply cannot be represented accurately as a combination of two base colours. High-contrast edges, such as text or hard boundaries between objects, have very distinct colours, leading to a significant loss in quality. This can be seen in textures compressed in older formats, such as S3 Texture Compression (S3TC). Figure 1 contains an image of a Firefox taskbar icon exactly as one would see on a computer monitor on the left, compared against a compressed version of the same image. The quality loss due to the high-contrast edges can be clearly seen [Det+].



(a) Original (b) DXT1

Figure 1: Small Firefox icon compressed with S3TC (DXT1)

Modern formats such as BC7 [Khr, p. 190] and ASTC have support for block partitioning [Khr, pp. 256–258]. This allows a compressed block to contain multiple endpoints, where each texel contains both weight and endpoint assignment information. This improves the encoding error for such blocks. In ASTC, the partition assignments are not precomputed or stored explicitly, but rather procedurally generated by a hash function (see Appendix A.4). When partitioning modes are enabled, a 10-bit number is stored in the block, which is used as the seed for said hash function, which assigns texels to partitions in the decoding stage. Figure 2 demonstrates how a sample block could be encoded with partitioning.

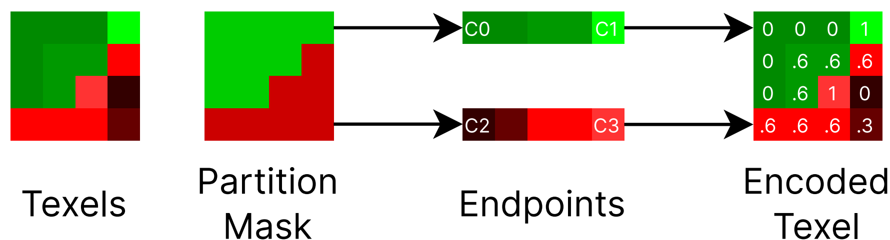


Figure 2: Partitioning Example

1.2.2 Integer Sequence Encoding

Bounded Integer Sequence Encoding (BISE) allows for the efficient packing of fractional bits for increased accuracy. With standard binary encoding, it is only efficient to represent ranges that correspond to the number of bits required to store them. In other words, a range of 0-15 may be represented with exactly 4 bits, while a range of 0-11 requires either a quantization down to 3 bits (0-7), which reduces quality, or storage within 4 bits, leading to wasted bits that could have been used elsewhere to achieve higher quality. To allow for increased flexibility in the encoder's choice of ranges to use for colour and texel weight data, ASTC defines trits and quints, which can be used to achieve fractional bits.

Trits and quints represent base-3 and base-5 values respectively. The trits or quints are interpreted as the upper bits (MSB) of the stored value, with the remaining bits as the lower bits (LSB). A representation of 35 with range 40 would thus be `4 011`. Five trits are packed together into $\lceil \log_2(3^5) \rceil = 8$ bits, while three quints are packed into $\lceil \log_2(5^3) \rceil = 7$ bits. This allows for an effective utilization of 1.6 and 2.33 bits per value respectively with $<$1% wasted storage. This allows for up to two additional ranges between each binary-efficient range. For example, between range 0-15 (4 bits) and 0-32 (5 bits) the encoder has the option to BISE-encode 0-19 or 0-23 ranges, freeing bits for other data. All combinations are shown below in Table 1.

Encoding \ States	**2**	3	**4**	5	6	**8**	10	12	**16**	20	24	**32**	40	48	**64**	80	96	**128**	160	192	**256**
Bits	1		2		1	3	1	2	4	2	3	5	3	4	6	4	5	7	5	6	8
Trits		1			1			1			1			1			1			1	
Quints				1			1			1			1			1			1		

Table 1: Efficient BISE Encodings

When encoding both bits and a trit/quint, the bits of the trit/quint are interleaved between individual values [Khr, p. 239]. This intentional design covers cases in which the number of values to be stored is not a multiple of 3 or 5. The number of trit/quint bits to be stored in total following the value corresponds to $\lceil \log_2(3^n) \rceil$ for trits and $\lceil \log_2(5^n) \rceil$ for quints, where n is the number of values to be stored within the group. The trit/quint values are then interleaved based on the number of additional bits required to store the new trit/quint states. For instance, the storage of one quint value requires $\lceil \log_2(5^1) \rceil = 3$ bits, and thus 3 bits representing the quint follow the number of lower bits required for the range (cf. Table 1). Storing two quints requires $\lceil \log_2(5^2) \rceil = 5$ bits. However, since 3 bits have already been set alongside the first value, only two bits must follow the second value. Since the range, number of weight points, and colour values are defined separately in ASTC, the decoder knows the number of expected values ahead of unpacking such that invalid data is not read and no padding is required.

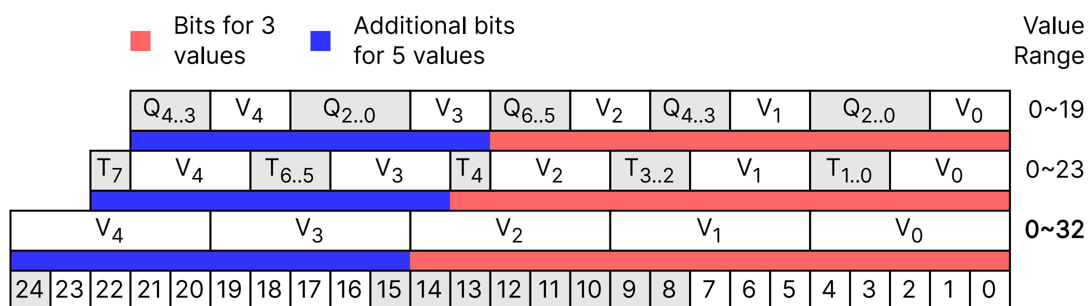


Figure 3: Efficiency of various BISE encodings

As Figure 3 shows, encoding only one or two values (shown as V_i) and their corresponding trits or quints (T_i and Q_i) with BISE may be inefficient. However, it is usually the case in ASTC that there are significantly more than two values to group together for encoding. For instance, the aforementioned colour endpoints in Section 1.2.1 consist of 6 different values. It can also be seen in the example for range 0-19 that each group of 3 quints (or 5 trits) are indepdendent of each other.

1.2.3 Bit Allocation

The number of bits available per weight and colour channel depends on the selected options [Khr, pp. 234–237]. Adding more partitions in ASTC is not free; colour and weight resolution suffers as more partitions are added. When compressing a 4x4 block with a single partition (i.e., one pair of base colours per block), 17 bits are used for metadata (block and colour modes). This leaves 111 bits available to the compressor to allocate between colour and weight data. Encoding the RGB values directly consumes 48 bits, leaving 63 bits for weight information (3.9 bpt). The closest available range is 0-11 or ≈3.58 bpt. Conversely, if maximum weight resolution provides the most optimal encoding, 31 bits remain for colour representation, or ≈5.16 bits per channel. The closest available range is 0-31, or 5 bits per texel. The stored bits are interpreted as normalized integer values: for example, the value 15 stored within 4 bits represents the maximum colour or weight intensity. The encoder can decide at runtime, which tradeoffs between colour and weight bits make sense for a particular block.

1.2.4 Block Layout

In contrast to other texture formats, which often provide a single block layout, ASTC provides several layouts for different types of textures [Khr, p. 234]. For this compression program, only the RGB LDR and void extent block types are of concern. The colour data immediately follows the configuration data, whereas the weight data starts from the most significant bit, growing downwards. Below is a sample bit layout for a single-partition block encoding. Note that the trits are sequentially numbered in the example for brevity, but they are not all dependent on each other (cf. BISE packing in Section 1.2.2).

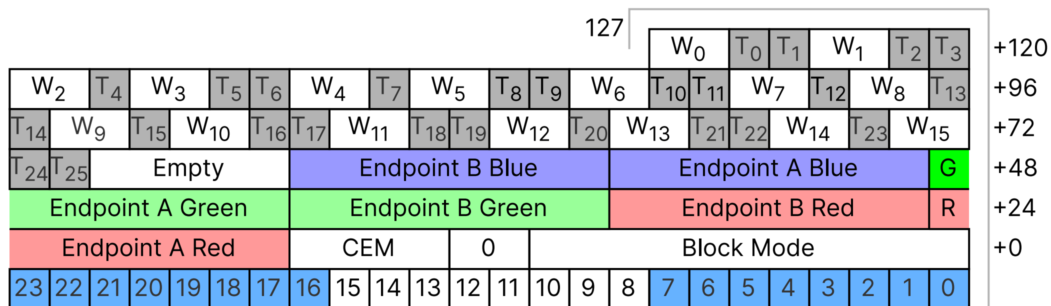


Figure 4: Sample 1-partition block layout

One can also see the benefit of BISE in Figure 4; the number of wasted bits using BISE range 12 is lower than if 3 bits per weight (range 8) were used, and full colour resolution can still be maintained.

Blocks which consist of a single solid colour are encoded as "void-extent" blocks [Khr, p. 259]. These blocks only contain the colours of a single endpoint at full resolution and no weight information, and may describe which neighbouring blocks also contain this solid colour (i.e., the "extent" of the colour). However, the latter is not strictly necessary and primarily exists to reduce memory accesses for hardware implementations. The bit layout is shown in Figure 5, where the s and t coordinates refer to the horizontal and vertical positions respectively.

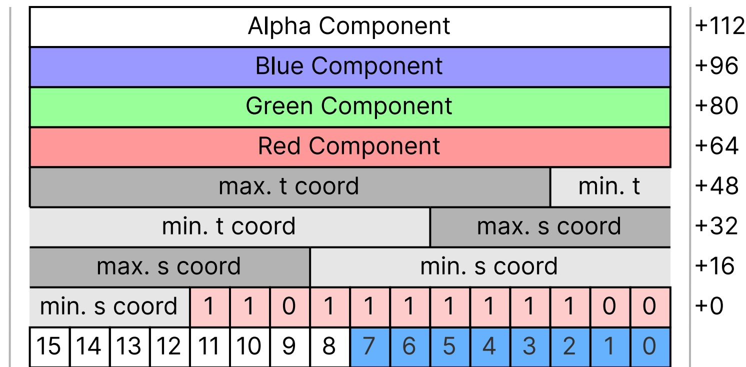


Figure 5: Sample void-extent block layout

1.3 ASTC Compressor

An ASTC compressor should be able to perform the previously mentioned operations in a reasonable timeframe. "Reasonable" is an arbitrary target, defined here as a 1080p image requiring less than a minute to encode (i.e., approximately 35,000 pixels per second per thread). The compressor application described in this report consists of two components: libmpastc, which is the underlying library capable of ASTC texture compression, and the mpastc user-facing application, which provides a command-line interface and user-specified image import capabilities. As the goal of this program is primarily to serve as a proof of concept, the number of ASTC features supported is limited. In particular, for ASTC-formatted export, the compressor is only capable of generating blocks with a specific combination of colour and weight bits. However, the encoder is capable of performing all necessary calculations to determine the best quantization level, and export its own intermediate values as a regular PNG-formatted image. The compressor also supports only RGB (red-green-blue, 8 bits per colour channel in little-endian order) data, both as the input and output.

To improve wall-clock runtimes, the compressor also supports the parallelization of ASTC compression steps over multiple threads with OpenMP and multiple physical nodes with Message-Passing Interface (MPI). The program consists of a serial implementation and two independent OpenMP and MPI parallelization layers. The report further details the serial and parallel implementation of the mpASTC program, as well as the specific techniques used in the compression pipeline. In addition, it also covers the performance and scalability with different levels and combinations of parallelization methods.

2 Serial Implementation

The serial implementation consists of the functions required to convert a raw block of pixels from a user-provided buffer or image into an ASTC block. This consists of colour endpoint selection, partitioning, and quantization steps. These steps are detailed in this section.

2.1 Intermediate Value Conversion

The compressor supports 3-channel, 8-bit-per-channel input. However, working directly on this input data would lead to a loss of accuracy due to rounding errors during compu-

tation as a result of the limited bit count. The values of each channel are thus converted into 32-bit floating-point values and copied to a temporary buffer, which all computations in the following implementation work with. This increases the accuracy of the final encoding and makes weight calculations especially simpler, since the computed output varies between 0 and 1. The exact code for this process can be found in the block extraction function, in Listing 11.

2.2 Endpoint Calculation

Two methods of endpoint selection were evaluated in mpASTC based on the astcrt paper [Oom]: Principal Component Analysis (PCA) and simple min/max endpoint selection. PCA is a dimensionality reduction technique that, in this context, reduces the array of vectors represented by the texels in a block in RGB colour space to a single vector representing the direction of maximum variance across the different texels. The min/max method simply finds the lowest and highest sum of RGB texel values in the block, setting these to the endpoint values. While PCA is slower, it is possible to find a better solution MSE-wise relative to the min/max method alone. In testing, however, PCA was sensitive to outliers, often producing results that, while containing a reasonably low Mean Squared Error (MSE), looked subjectively worse than endpoint values obtained by the much simpler and faster min/max method. Thus, the latter method was ultimately chosen for endpoint selection.

After the initial endpoint selection, weights are calculated by projecting the individual texels onto the vector represented by the difference between the start and end colours of the endpoint, as described below in Equation 2. This produces a weight value between 0 and 1, which is further used in the quantization steps.

$$\begin{aligned} \vec{E_\Delta} &= \vec{E_b} - \vec{E_a} \\ \vec{T_\Delta} &= \vec{T_i} - \vec{E_\Delta} \\ P_i &= \frac{\vec{T_\Delta} \cdot \vec{E_\Delta}}{\vec{E_\Delta} \cdot \vec{E_\Delta}} \vec{E_\Delta} \\ W_i &= median(0, \frac{||P_i||}{||E_\Delta||}, 1) \end{aligned} \tag{2}$$

where:

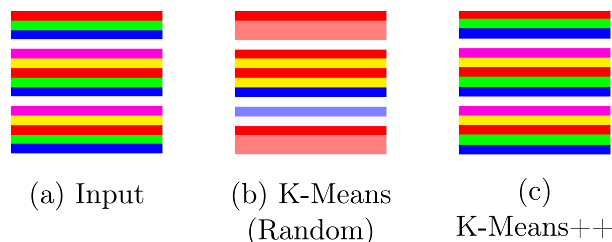
- Vector elements represent colours in RGB space (i.e., Red=x, Green=y, Blue=z)
- $\vec{E_b}$ and $\vec{E_a}$ are the colour endpoints
- $\vec{T_i}$ is the vector representing the RGB values of the texel at position i within the block
- W_i is the calculated weight for texel i.

Void extents are used when blocks are solid. However, only endpoint data (no extent information) are populated, avoiding dependencies between blocks that would significantly increase the implementation complexity and reduce parallelism for images that would make heavy use of these. It was observed that images such as digital illustrations in simpler art styles result in void extent blocks being encoded more often relative to "natural" images such as those of landscapes.

2.3 Partitioning

Two partitioning methods were evaluated. The first consisted of performing full endpoint calculations on every possible partitioning available in ASTC. This method produces the best quality images, however, the performance was unusably slow. When this was evaluated with an early version of mpASTC, this partitioning method achieved between 300-1000 pixels per second, per thread. For context, this would result in a 1080p image taking as long as two hours to encode on a single thread.

Instead, the method used in the final implementation is one described in astcrt [Oom], with certain algorithmic changes designed to improve encoding error for specific types of images. Namely, for every partition count (2, 3, and 4), k-means++ is first used to generate high-quality seed values [AV07], which are provided as the initial centroids to a common K-means algorithm (Lloyd's Algorithm) [Llo82]. This produced an ideal partitioning for images with sharp changes in colour far more often relative to k-means initialized with random values (either in RGB space or selected from the texel data). Note that since k-means++ and Lloyd's algorithm are heuristic methods, there is no guarantee that every encoding pass will produce the ideal clustering of pixels. Figure 6 shows an extreme scenario consisting of sharp colour changes, and how regular k-means often fails to encode this block properly.



(a) Input (b) K-Means (Random) (c) K-Means++

Figure 6: Problematic block encodings

The number of valid partitions and their layouts are also pre-computed, as the implementation of partition pattern generation is designed for hardware [Khr, pp. 256–257]. In C, the reference code provided by Khronos (see Appendix A.4) is rather complex and time-consuming to run. In addition, because the partitions are procedurally generated, not all of them are useful. Thus, the following partitions are excluded, and example blocks are provided in Figure 7 to illustrate the rejected blocks.

- Solid partitions
- Exactly identical to previously observed partitions
- Functionally identical to previously observed partitions (same structure, different partition assignment)
- Fewer partitions than expected (e.g. 3 active partitions in 4-partition layout)

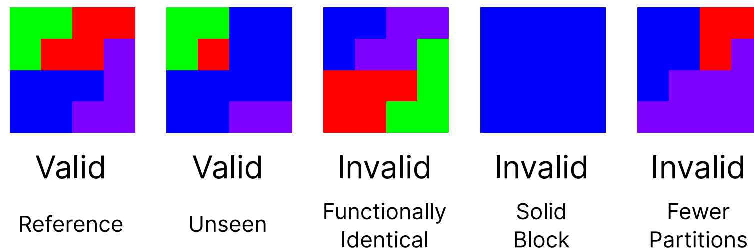


Figure 7: Sample 4-partition blocks

ASTC allows for up to 1024 different partitionings per partitioning level (i.e., 1024 n-partition configurations per $n \in [2, 4]$). By applying the previous filtering criteria to these partition configurations, there are 437, 329, and 321 valid 2-, 3-, and 4-partition configurations remaining respectively. This reduces the search space size approximately a third of the original. The valid n-partition mappings are then concatenated into a contiguous array. A reverse lookup table is used to map these contiguous partition mappings to the actual ASTC partition identifier. For example, 2-partition configuration ID 436 maps to the actual ASTC 2-partition ID 1023.

2.4 Quantization

All possible combinations of weight and colour bit allocations for a given partition count are pre-computed and stored in a lookup table. Each combination is tested by downsampling the calculated weight and colour values to a specified resolution, and the combination that produces the lowest MSE is selected. The possible encodings for different partition counts are shown in Listing 1, and 255 is used to denote the end of the array:

```
const uint8_t enc_table_4x4_1[] = {
    QUANT_32,  QUANT_32, QUANT_64, QUANT_24, QUANT_96, QUANT_20, QUANT_192, QUANT_16,
    QUANT_256, QUANT_12, 255,      255};
const uint8_t enc_table_4x4_2[] = {
    QUANT_10, QUANT_12, QUANT_12, QUANT_10, QUANT_16, QUANT_8, QUANT_24, QUANT_6,
    QUANT_32, QUANT_5,  QUANT_40, QUANT_4,  QUANT_64, QUANT_3, QUANT_96, QUANT_2,
    255,      255};
const uint8_t enc_table_4x4_3[] = {
    QUANT_8,  QUANT_6, QUANT_10, QUANT_5, QUANT_12, QUANT_4, QUANT_16, QUANT_3,
    QUANT_24, QUANT_2, 255,      255};
const uint8_t enc_table_4x4_4[] = {
    QUANT_8, QUANT_3, QUANT_10, QUANT_2, 255, 255};
```

Listing 1: Available quantization levels

In Listing 1, two successive elements represent a colour and weight quantization level respectively. For example, `QUANT_10, QUANT_12` in `enc_table_4x4_2` refers to a quantization level corresponding to 10 states per colour channel and 12 states per texel weight respectively. When `BYPASS_ENC_COMB_CHECK` is defined (by default), only a limited subset of these combinations is used. This is currently necessary to produce valid ASTC output. The combinations used are as follows:

```
const uint8_t enc_s_table_4x4_1[] = {QUANT_32, QUANT_32, 255, 255};
const uint8_t enc_s_table_4x4_2[] = {QUANT_16, QUANT_8, 255, 255};
const uint8_t enc_s_table_4x4_3[] = {QUANT_16, QUANT_3, 255, 255};
const uint8_t enc_s_table_4x4_4[] = {QUANT_8, QUANT_3, 255, 255};
```

Listing 2: Preset quantization levels

For the actual encoding of trit and quint values, a table-based method is recommended in the Khronos Data Format Specification [Khr, p. 240]. Quints and trits are packed using the tables from the ARM reference ASTC encoder [ARMa], which takes a group of 3 and 5 values to be encoded for quints and trits respectively, and outputs a 8-bit value corresponding to the trits/quints of the given values. The bits are then written individually to the output stream in the order detailed in Section 1.2.2. For endpoint values, the weights are written in little-endian order, while both bit and byte orders are flipped when writing weight data in line with the ASTC specification.

3 Parallel Implementation

The parallel implementation includes parallelization with OpenMP for intra-node and MPI for inter-node communications. It builds upon the aforementioned serial implementation by adding the ability to distribute ASTC blocks to be compressed across multiple processors.

As the choice was made not to fill "extent" information of void-extent blocks, ASTC blocks can now be considered independent from the compressor's perspective. Work can thus be distributed at a block level without much concern for data races. Parallelization beyond this level was not further explored, as the relatively low measured runtime of each block compression (<1 msec/block) would likely have been far exceeded by the overhead and complexity of communication. In addition, the number of blocks available for distribution far exceeds the number of available threads: for a 1080p image, there exist 129,600 unique 4x4 blocks. The processor utilization can therefore remain high without any further modifications.

3.1 MPI/OpenMP Hybrid Implementation

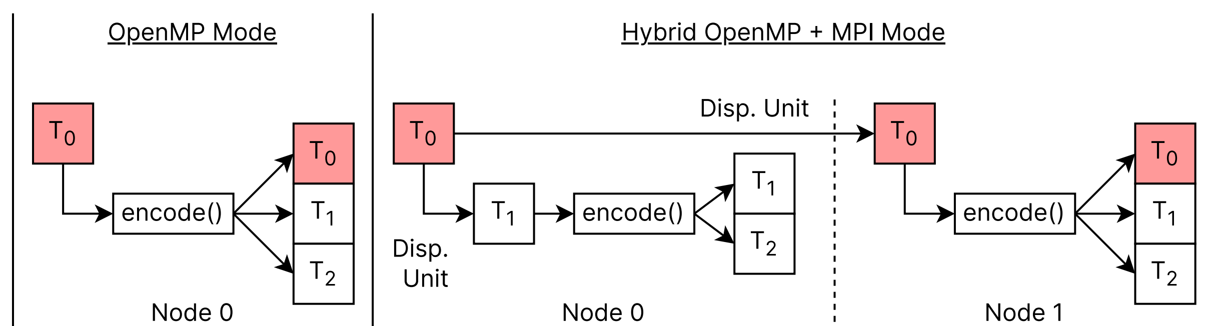


Figure 8: Parallel Implementations of mpASTC

As shown in Figure 8, two distinct threading implementations exist. The core library, libmpastc, contains all codepaths for compression and multi-threading with OpenMP. The mpastc user-facing application, however, is capable of spawning regular pthreads and managing MPI communications. To interoperate with OpenMP, mpastc requests a specific thread count be used for block encoding from libmpastc when calling its encoding function. With MPI disabled, either at compile- or run-time, mpastc instructs libmpastc to compress groups of blocks with the number of threads specified at runtime (or by default, all available threads on the system).

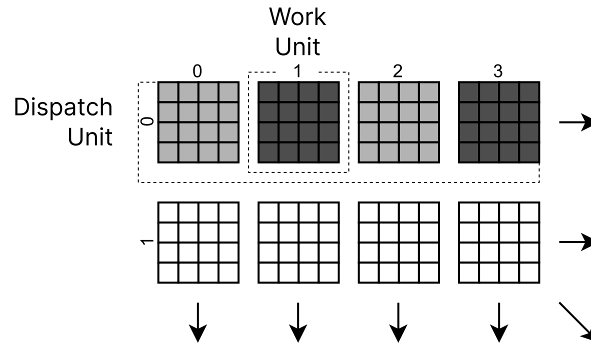


Figure 9: Work/Dispatch Unit Layout

In either operating mode, each block is identified internally with a unique identifier based on the order of appearance in row-major order as shown in Figure 9, and these IDs are statically allocated to threads implicitly with OpenMP. The rationale for the static thread allocation is that, for complex images, the overhead of dynamic scheduling is avoided: blocks with many colours take a similar amount of time to encode relative to each other. This is at the expense of images containing both high and low detail areas, however, it was decided that the trade-off was acceptable.

The identifier of each block is used to determine the location in memory of the block to be extracted. Each ID is translated to an independent chunk of memory containing a 4x4 block, shown in Listing 3 below. The functions are expanded and further explained in Listing 11.

```
1  #pragma omp parallel for num_threads(n_threads)
2  for (uint32_t i = 0; i < meta[META_WIDTH] * meta[META_DISPATCH] / bs; i++) {
3      /* setup code omitted */
4      /* i = index, bw/bh = block width/height, x/y = output coordinates */
5      id_to_index(i, w, bw, bh, &x, &y);
6      /* buf = input image data, ir_data = working buffer, 1 = convert to floats */
7      extract_block((uint8_t *)buf, ir_data, 1, w * 3, x, y, bw, bh);
8      /* computation code omitted */
9  }
```

Listing 3: OpenMP Work Distribution

Communication is performed over shared memory between the control and worker threads on Rank 0, and over MPI for remote nodes. The first thread of Rank 0 is always the dedicated control thread, responsible for all communication. Instead of sending individual blocks over the network, which would increase the impact of network latency on performance, blocks are grouped into dispatch units. These units span the entire width of the image and are an integer multiple of the block height, which makes memory accesses to transfer these blocks fully contiguous.

The ideal height multiplier depends on the number of threads available on the systems in use. If, for instance, the image width is 256 pixels, there are 64 blocks per dispatch unit. In a system with more than 64 threads, this results in no useful work being scheduled on the remaining threads. To compensate for this, the dispatch height can be increased to, for instance, twice the block height. In that case, the work units may be efficiently distributed across 128 threads.

Within a dispatch group, each block is identified by its position. In line with the ordering of ASTC blocks, they are in row-major order, with the top-left block being index 0 and the bottom-right block being index n - 1, where n is the number of blocks in the

work unit. Since each thread receives independent block IDs, data races are avoided. Unlike the pure OpenMP-based threading implementation, the block IDs are only unique within a work unit, and each work unit contains a unique identifier. For example, the first block of the dispatch unit 0 received by rank 1 is ID 0, and the first block of the dispatch unit 1, received by rank 2, is also ID 0.

3.2 Local Worker

The work distribution pattern is simple, and thus communicating over MPI on the root node to other threads would lead to unnecessary overhead. Instead, a separate thread is split off the main thread and designated as a worker thread, as shown in Figure 8. The worker and controller thread communicate through a shared structure containing the data required to compress a block.

```
1  struct local_worker_opts {
2      struct mpa_options * opts;    // Global app options (user-specified)
3      std::atomic<int64_t> counter; // Progress counter / lock
4      uint32_t * meta;              // Metadata
5      void * in_buf;                // Raw texel buffer
6      void * out_buf;               // Output buffer
7      int nthreads;                 // Number of threads to use
8  };
```

Listing 4: Local worker control structure

Locks are implemented by means of a simple atomic counter, which keeps synchronization tasks in userspace. Whenever the counter is even, the controller can modify the structure, and when odd, the worker can. When done, control is handed off by incrementing the counter. Alternatively, if the counter is set to -1 (there is no more work to do), the local worker terminates. The only variables changed in each iteration are the pointers to the input and output buffer, which point to the starting position of an unassigned dispatch unit, in the same shared queue as the MPI workers. No texel data is copied. The number of threads is set to one less than the number of available threads on the system, preventing contention with the main communication thread. These threads are then created by OpenMP in the local worker function.

3.3 MPI Workers

Before work is possible, all participants receive metadata about the image to be compressed from the root node. This is a flat array of 32-bit integer values containing the image's width, height, pixel pitch (bytes per pixel), row stride (bytes per line), dispatch (work unit) height, input format, output format, block width, block height, option flags, and thread count limits. With this information, worker nodes may allocate the memory required to store input and output buffers. As the only supported conversion is 24-bit RGB to 4x4 ASTC, the input and output formats are currently ignored.

Communication is double-buffered with tasks being distributed in a round-robin fashion. Nodes with no work assigned (i.e., two empty buffers) take priority over nodes with free receive slots (one empty buffer). Both the controllers and workers use non-blocking MPI functions. The memory regions for each dispatched block are spaced such that there is no overlap: the controller ensures each specific range of texels, identified by a unique sequential integer ID, is independent such that no local memory copies are required. To

ensure that messages reach the correct buffer, the front and back buffers use separate MPI tags. An example of assignments of buffers to ranks and threads is shown in Figure 10.

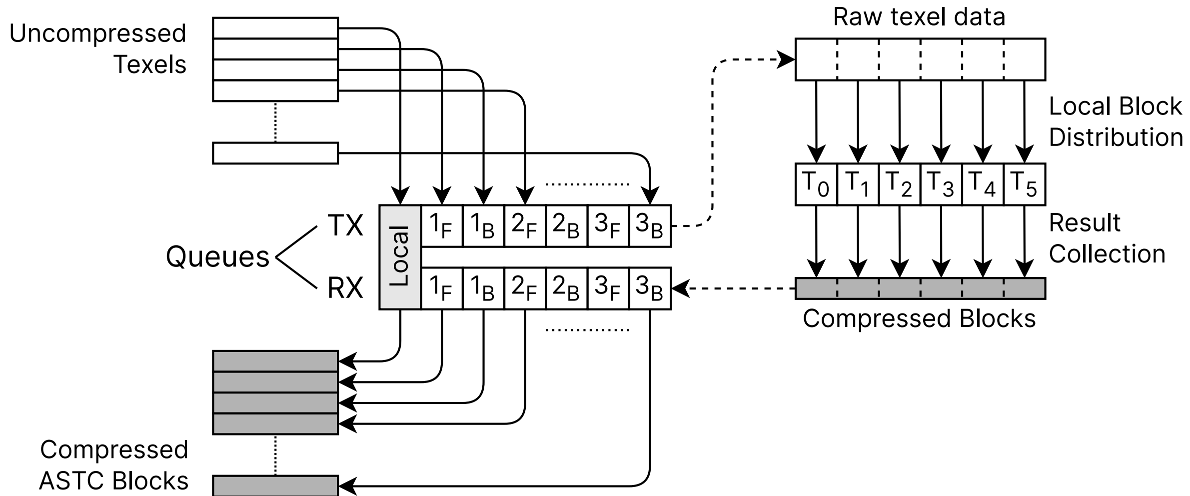


Figure 10: MPI/Local Worker Buffer Allocation

When there are no more blocks to submit to workers, the controller sends a one-byte message to all nodes with a specific tag number, defined in the codebase as `MPA_TAG_CTRL`. Once any message with this tag is received by the MPI worker, allocated resources are cleaned up and the worker process terminates.

4 Performance

To gauge the performance of mpASTC, benchmarks were run on High-Performance Computing (HPC) clusters with both high thread counts and high-speed networking to gauge the horizontal and vertical scalability limits of the program.

4.1 Configuration

On the Ampere (1x Q80-30) and Intel (2x Xeon 9242) nodes, mpASTC was compiled with optimization enabled, debug checks disabled, and quantization error tests enabled. Only the runtime and pixel rates are printed to standard output at the end of the test run (benchmark mode `-b`). The runtime between image load completion and the root node receiving all compressed results was measured. File I/O is excluded from calculations as this is dependent mainly on the underlying filesystem, which is not the topic of interest especially on a shared cluster filesystem: saving times vary between <1 second to multiple seconds depending on uncontrollable neighbour utilization and severely reduces the interpretability of the results. The application was additionally run on both systems with exclusive access to the node, to avoid noisy neighbours as much as reasonably possible.

Multi-node MPI was unavailable on the ARM platform. As such, the benchmarks were performed on the GWDG SCC medium partition [2] with the Intel OneAPI compiler (2022.0.1) and Intel OneAPI MPI (2021.5.0). The Omni-Path Remote Direct Memory

[2] See Appendix A.1 for details

Access (RDMA) transport is enabled by default. OpenMPI on the GWDG SCC is currently unstable, and as such will not be compared. The exact command sequences used for testing are shown in Appendix A.3.

In thread-only benchmarks with MPI disabled, all threads as specified on the command line are used- there is no distinction between worker and control threads as this is handled by OpenMP itself. In MPI benchmarks, the thread count represents the sum of active threads across all nodes. However, one thread on rank 0 is always the dedicated control thread. In other words, two nodes with two threads per node is shown in the chart as 4 threads, where 3 are in use for encoding and 1 for control.

All benchmarks encoded the same image, a full-resolution image of the Carina Nebula taken by the James Webb Space Telescope [Nat], cropped to 14572x8440 to fit the 4x4 block size. This image was chosen due to its high resolution (123 megapixels), level of detail, and permissive license.

4.2 Threading Performance

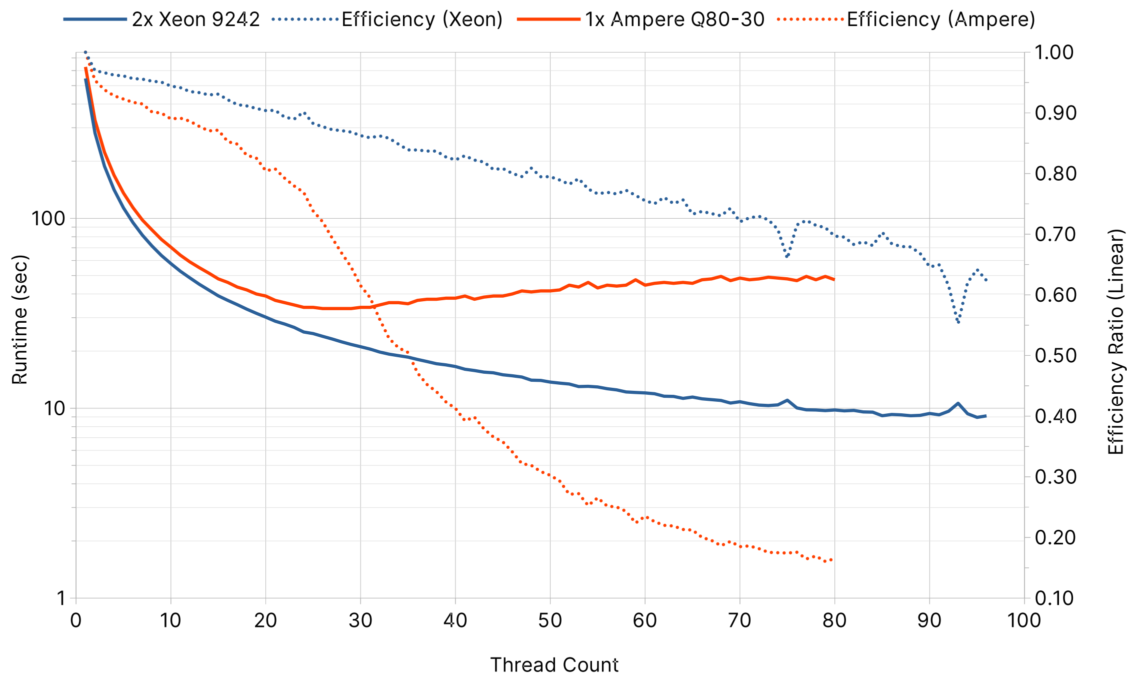


Figure 11: Threaded Compressor Performance

In Figure 11, runtime is compared against thread count. Efficiency refers to the runtime of the threaded implementation multiplied by the thread count, relative to the reference single-threaded implementation. For instance, a runtime of 2 seconds on 2 threads relative to 1 second on 1 thread represents an efficiency of 100% (1.0). The runtime on both Intel and Ampere systems is initially inversely proportional to the number of threads used. However, a performance regression is observed after 25 threads on the Ampere system and a linear drop in threading efficiency was observed on the Intel system. This suggests that the workload is heavily cache- and memory-bound: the Ampere system contains less cache (Ampere: 1MB/core, Intel: 2.5MB/core [3]) and fewer memory channels (Ampere:

[3] L2 + L3 cache, excluding L1. Gathered from lstopo, see Appendix A.6

8, Intel: 12) than the Intel system [Amp; Int].

Results on the Intel system suggest that the workload itself lends itself well to parallelism, as the implementation requires little to no additional data movement relative to a single-threaded implementation. The largest contributor to the efficiency loss, therefore, is most likely the aforementioned limited memory bandwidth. This is further explored in the MPI performance tests, which also produced positive results.

4.3 MPI Performance

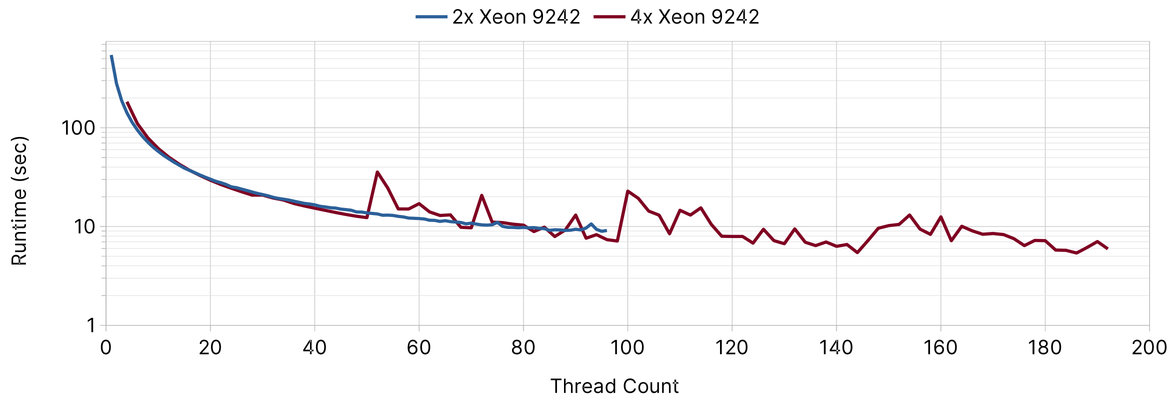


Figure 12: Initial 2-Node Results

Initial results shown in Figure 12 with MPI enabled clearly show erratic performance past 24 threads per node (48 threads total). Unlike with the threading implementation, remote nodes must first wait for incoming data over the network and thus cache prefetching may be limited. This is likely further compounded by the fact that the network adapter resides only on one of the Non-Uniform Memory Access (NUMA) nodes of the system; if the network adapter NUMA node does not match the NUMA nodes of the threads performing the computation, latencies and bandwidths are further increased and decreased respectively. This is the most likely conclusion on the Intel system, as the Xeon 9242 contains 24 cores per NUMA node, two NUMA nodes per socket, and two sockets per system (cf. Listing 9).

Multi-node operation producing faster results with higher efficiency, relative to the same number of threads on a single node, can be observed after 16-20 threads. Performance peaks can be seen close to integer multiples of the NUMA node size. To observe possible speedup with an optimized configuration, benchmarks were re-run and only runtimes of "ideal" configurations: 24, 48, and 96 threads, multiplied by the number of nodes for the total thread count, were included.

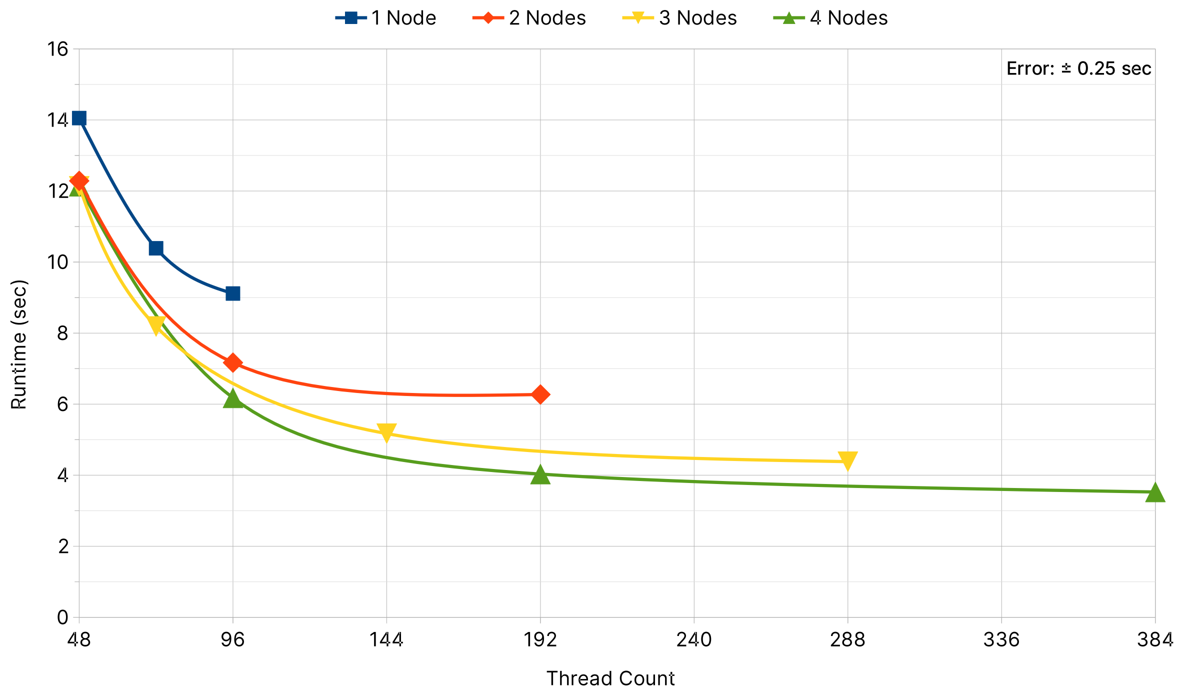


Figure 13: Runtimes for Various Node Counts

When only these configurations are observed, there is a good level of performance improvement across all node counts, shown in Figure 13.

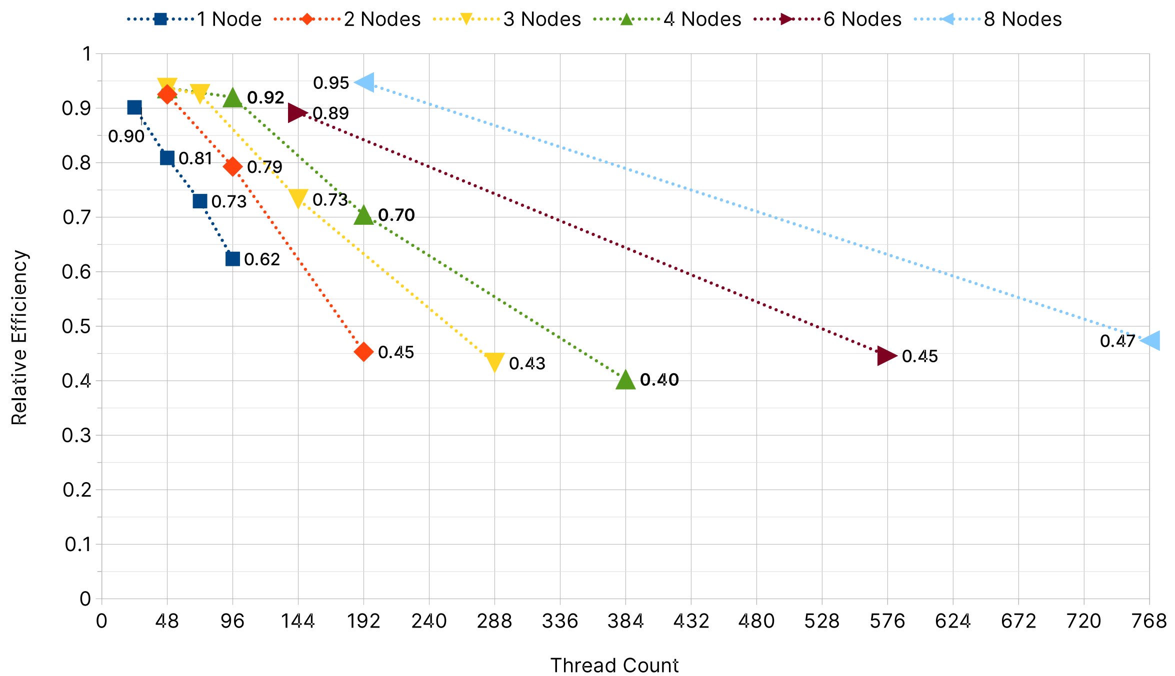


Figure 14: Relative Efficiency for Various Node Counts

Figure 14 shows the efficiency relative to a single-threaded implementation, as defined in Section 4.2. From these results, it can be observed that the MPI layer adds little

overhead relative to the performance improvements that can be gained through the increased amount of cache on multiple nodes. Peak efficiency can be achieved when only one NUMA node per physical node is used: for all configurations this can be seen in the band between 90 and 100% efficiency. Likewise, all multi-node configurations exhibit the same efficiency loss at full utilization: they are grouped together in the 40-50% band. The variance is most likely a side effect of the performance variation and background task noise of individual nodes, for which limited control is available: node selections were not always the same as they are based on availability, and background jobs are unpredictable.

5 Conclusion

ASTC is a high-quality, but slow to encode compressed texture format. In order to improve encoding times, a new ASTC compressor program (mpASTC) was created, and parallelization techniques were used to efficiently split work across both multiple threads on the same node, and multiple different nodes. Parallel encoding speeds have been improved substantially relative to a single-threaded implementation. In addition, due to the nature of the ASTC format's independent blocks, the memory-intensive nature of the methods used within the mpASTC compressor, and the use of non-blocking double-buffered RDMA communication, the low overhead of inter-node communication on a fast network is significantly outweighed by the increased amount of aggregate memory bandwidth available on multiple physical machines. Efficiency reaches over 90% relative to a single-threaded implementation with 8 nodes simultaneously active, suggesting that headroom still exists for parallelization beyond the 768 threads tested.

Gathered benchmarking results indicate that the parallelization of block-based texture compression formats could also be performed efficiently using techniques in, or similar to, those used in mpASTC. Time-sensitive applications, such as game development, could benefit from parallelized compression techniques to decrease asset preparation time, especially in the prototyping stages.

References

[Amp] Ampere Computing LLC. *Ampere Altra Family Product Brief.* https://amperecomputing.com/briefs/ampere-altra-family-product-brief.

[ARMa] ARM Limited. *ARM ASTC Encoder.* https://github.com/ARM-software/astc-encoder.

[ARMb] ARM Limited. *ASTC Benefits.* https://developer.arm.com/documentation/102162/0430/ASTC-benefits.

[ARMc] ARM Limited. *The ASTC Format.* https://developer.arm.com/documentation/102162/0430/The-ASTC-format.

[AV07] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. ISBN: 9780898716245.

[Cha] David Chait. *Using ASTC Texture Compression for Game Assets.* https://developer.nvidia.com/astc-texture-compression-for-game-assets.

[Det+] Tim Dettmar et al. *Telescope Project / Uncompressed video streaming over local networks for real-time interactive content.* https://public.cdn.timd.io/TelescopeProjectThesisPub.pdf.

[Int] Intel Corporation. *Intel® Xeon® Platinum 9242 Processor.* https://www.intel.com/content/www/us/en/products/sku/194145/intel-xeon-platinum-9242-processor-71-5m-cache-2-30-ghz/specifications.html.

[Khr] Khronos Group. *Khronos Data Format Specification version 1.3.1, April 2020.* https://registry.khronos.org/DataFormat/.

[Llo82] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[Nat] National Aeronautics and Space Administration (US). *Carina Nebula (High resolution).* https://www.flickr.com/photos/nasawebbtelescope/52259221868/in/album-72177720300469752/.

[Oom] Daniel Oom. *Real-Time Adaptive Scalable Texture Compression for the Web.* https://hdl.handle.net/20.500.12380/234933.

A Appendix

A.1 Acknowledgements

This work used the Scientific Compute Cluster at GWDG, the joint data center of Max Planck Society for the Advancement of Science (MPG) and University of Göttingen. The "medium" partition used in benchmarks consists of dual-socket Xeon 9242 nodes with 384 GB of RAM and 100 Gbit/s Omni-Path networking, running Scientific Linux 7.9. For more information related to the GWDG SCC, visit `https://gwdg.de/en/hpc/`. ARM benchmarks were conducted on the GWDG Future Technologies Platform (FTP).

A.2 Source Code Repository

The source code repository can be found at `https://gitlab.gwdg.de/tim.dettmar/mpastc-archive`. Note that this is a snapshot of the codebase at the time of writing, and may not be the latest version of mpASTC. The repository also includes the raw benchmark results.

A.3 MPI Benchmark Commands

```
# Setup
export OMP_PLACES=sockets OMP_PROC_BIND=close
module load intel-oneapi-compilers intel-oneapi-mpi
salloc -p medium -N <nodes> -c 96 --time=01:30:00
ssh <node>
    cmake -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx -DENABLE_MPI=1 .
    make -j96
exit

# Single-partition ASTC export test (functionality test)
# Skip if you don't have the ARM ASTC encoder (astcenc)
./mpaenc --file carina.jpg --out carina.astc -t <threads>
astcenc -dl carina.astc carina.png

# Single-node bench
ssh <node>
    for i in {1..96}
    do
        srun -N <nodes> -n <nodes> -c $i \
            ./mpaenc --file carina.jpg --out carina.png -b -d -p -t $i
    done
exit

# Multi-node bench
for i in {2..96}
do
    srun -N <nodes> -n <nodes> -c $i \
        ./mpaenc --file carina.jpg --out carina.png -b -d -m -p -t $i -s 10
done
```

Listing 5: MPI Benchmark Commands

A.4 ASTC Partition Generation

The code is from the Khronos Data Format specification, with some slight modifications [Khr, pp. 256–258]. In essence, `select_partition` takes the partition index (seed), hashes the seed value using the `hash52` function, and applies certain bit manipulations to it, returning an integer value between $[0, parts)$ indicating the partition number of the texel at coordinate x, y (coordinates are local to the block).

```
uint32_t hash52(uint32_t p) {
    p ^= p >> 15; p -= p << 17; p += p << 7; p += p << 4;
    p ^= p >> 5;  p += p << 16; p ^= p >> 7; p ^= p >> 3;
    p ^= p << 6;  p ^= p >> 17; return p;
}

uint8_t select_partition(int seed, int x, int y, int z, int parts, int num_texels) {
    if (num_texels < 31) {x <<= 1; y <<= 1; z <<= 1;}
    seed += (parts - 1) * 1024;
    uint32_t rnum = hash52(seed);
    uint8_t seed1 = rnum & 0xF;
    uint8_t seed2 = (rnum >> 4) & 0xF;
    uint8_t seed3 = (rnum >> 8) & 0xF;
    uint8_t seed4 = (rnum >> 12) & 0xF;
    uint8_t seed5 = (rnum >> 16) & 0xF;
    uint8_t seed6 = (rnum >> 20) & 0xF;
    uint8_t seed7 = (rnum >> 24) & 0xF;
    uint8_t seed8 = (rnum >> 28) & 0xF;
    uint8_t seed9 = (rnum >> 18) & 0xF;
    uint8_t seed10 = (rnum >> 22) & 0xF;
    uint8_t seed11 = (rnum >> 26) & 0xF;
    uint8_t seed12 = ((rnum >> 30) | (rnum << 2)) & 0xF;
    seed1 *= seed1; seed2 *= seed2;   seed3 *= seed3;   seed4 *= seed4;
    seed5 *= seed5; seed6 *= seed6;   seed7 *= seed7;   seed8 *= seed8;
    seed9 *= seed9; seed10 *= seed10; seed11 *= seed11; seed12 *= seed12;
    int sh1, sh2, sh3;
    if (seed & 1) {
        sh1 = (seed & 2 ? 4 : 5);   sh2 = (parts == 3 ? 6 : 5);
    } else {
        sh1 = (parts == 3 ? 6 : 5); sh2 = (seed & 2 ? 4 : 5);
    }
    sh3 = (seed & 0x10) ? sh1 : sh2;
    seed1 >>= sh1; seed2 >>= sh2;  seed3 >>= sh1;  seed4 >>= sh2;
    seed5 >>= sh1; seed6 >>= sh2;  seed7 >>= sh1;  seed8 >>= sh2;
    seed9 >>= sh3; seed10 >>= sh3; seed11 >>= sh3; seed12 >>= sh3;
    int a = seed1 * x + seed2 * y + seed11 * z + (rnum >> 14);
    int b = seed3 * x + seed4 * y + seed12 * z + (rnum >> 10);
    int c = seed5 * x + seed6 * y + seed9  * z + (rnum >> 6);
    int d = seed7 * x + seed8 * y + seed10 * z + (rnum >> 2);
    a &= 0x3F; b &= 0x3F; c &= 0x3F; d &= 0x3F;
    if (parts < 4) d = 0;
    if (parts < 3) c = 0;
    if (a >= b && a >= c && a >= d) return 0;
    else if (b >= c && b >= d) return 1;
    else if (c >= d) return 2;
    else return 3;
}
```

Listing 6: ASTC Partition Generation Code

A.5 Precomputing ASTC Partition Mappings

Running the function in Listing 6 with all combinations of $x \in [0, 4)$ and $y \in [0, 4)$ produces the actual mapping, which is then written to the output buffer in row-major order. The code to do so is as follows:

```
void gen_part(int bw, int bh, int parts, uint8_t * out) {
    int i = 0;
    for (int index = 0; index < 1024; index++) {
        for (int y = 0; y < bh; y++) {
            for (int x = 0; x < bw; x++) {
                out[i++] = select_partition(index, x, y, 0, parts, bw * bh);
            }
        }
    }
}
```

Listing 7: ASTC Partition Map Generation

The outputs are then filtered to remove identical or “useless” blocks as described in Section 2.3.

```
int reorder_blk(int bs, int parts, uint8_t * in, uint8_t * out) {
    /* Change the order by mapping partition ids based on order seen to
        cover cases like:

        0 0 0 1   1 1 1 0                   0 0 0 1
        0 0 0 1   1 1 1 0   both remapped   0 0 0 1
        0 0 1 1   1 1 0 0   to              0 0 1 1
        0 1 1 1   1 0 0 0                   0 1 1 1

    */
    uint8_t norm[bs];
    uint8_t map[parts];
    for (int k = 0; k < parts; k++) {
        map[k] = 255;
    }
    int ix = 0;
    for (int k = 0; k < bs; k++) {
        if (map[in[k]] == 255)
            map[in[k]] = ix++;
    }
    if (ix == 1)
        return 1;
    for (int k = 0; k < bs; k++) {
        out[k] = map[in[k]];
        assert(out[k] < 4);
    }
    return 0;
}

uint16_t filter_ident(int bw, int bh, int parts, uint8_t * data,
                      uint8_t * out) {
    int stat_solid = 0;
    int stat_ident = 0;
    int stat_fewer = 0;
    int stat_valid = 0;
    int bs = bw * bh;
    for (int i = 0; i < 1024; i++) {
        uint8_t * cand = data + i * bs;
```

```
39
40          uint8_t norm_cand[bs];
41          if (reorder_blk(bs, parts, cand, norm_cand) == 1) {
42              stat_solid++;
43              out[i] = 0;
44              continue;
45          }
46
47          /* A partition layout that has fewer actually used partitions will
48           * be discarded (e.g. a 4-partition layout only having two actual
49           * partitions mapped to texels) */
50
51          uint8_t obs = 0;
52          for (uint8_t j = 0; j < bs; j++) {
53              assert(norm_cand[j] < 4);
54              obs |= (1 << norm_cand[j]);
55          }
56
57          if (obs != (1 << parts) - 1) {
58              stat_fewer++;
59              out[i] = 0;
60              continue;
61          }
62
63          /* Whether to consider this candidate during partition selection */
64          uint8_t flag = 1;
65
66          for (int j = i - 1; j >= 0; j--) {
67
68              if (!out[j])
69                  continue;
70
71              uint8_t * cmp = data + j * bs;
72              uint8_t norm_cmp[bs];
73
74              if (reorder_blk(bs, parts, cmp, norm_cmp) == 1)
75                  continue;
76
77              uint8_t k;
78              for (k = 0; k < bs; k++) {
79                  if (norm_cmp[k] != norm_cand[k])
80                      break;
81              }
82
83              if (k == bs) {
84                  stat_ident++;
85                  flag = 0;
86                  break;
87              }
88          }
89
90          out[i] = flag;
91          if (out[i])
92              stat_valid++;
93      }
94      if (enable_print)
95          printf("Block: %dx%d, parts: %d, valid: %d, solid: %d, fewer: %d, "
96                 "ident: %d\n",
```

```
97                  bw, bh, parts, stat_valid, stat_solid, stat_fewer, stat_ident);
98
99      return stat_valid;
100 }
```

Listing 8: ASTC Partition Filtering

The output of these functions are then saved into a header file containing valid partition mappings. This is not included here, but can be found in the source tree under the directory `mpastc/codegen/partgen.c` where the above listings also reside.

A.6 System Topologies

The outputs have been trimmed to include relevant information.

A.6.1 Intel System

```
Machine (376GB total)
  Package L#0
    NUMANode L#0 (P#0 93GB)
      L3 L#0 (36MB)
        L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
        L2 L#1 (1024KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
        ...
        L2 L#23 (1024KB) + ... + PU L#23 (P#23)
    NUMANode L#1 (P#1 94GB) + L3 L#1 (36MB)
      L2 L#24 (1024KB) + ... + PU L#24 (P#24)
      ...
      L2 L#47 (1024KB) + ... + PU L#47 (P#47)
  Package L#1
    NUMANode L#2 (P#2 94GB)
      L3 L#2 (36MB)
        L2 L#48 (1024KB) + ... + PU L#48 (P#48)
        ...
        L2 L#71 (1024KB) + ... + PU L#71 (P#71)
      HostBridge L#5
        PCIBridge
          PCI 8086:24f0
            Net L#3 "ib0"
            OpenFabrics L#4 "hfi1_0"
    NUMANode L#3 (P#3 94GB) + L3 L#3 (36MB)
      L2 L#72 (1024KB) + ... + PU L#72 (P#72)
      ...
      L2 L#95 (1024KB) + ... + PU L#95 (P#95)
```

Listing 9: Intel System Topology

A.6.2 Ampere System

```
Machine (510GB total)
  Package L#0
    NUMANode L#0 (P#0 510GB)
    Die L#0 + L2 L#0 (1024KB) + L1d L#0 (64KB) + L1i L#0 (64KB) + Core L#0 + PU L#0
    (P#0)
    Die L#1 + L2 L#1 (1024KB) + L1d L#1 (64KB) + L1i L#1 (64KB) + Core L#1 + PU L#1
    (P#1)
    ...
    Die L#79 + L2 L#79 (1024KB) + ... + PU L#79 (P#79)
  ...
  HostBridge
    PCIBridge
      PCI 000c:01:00.0 (InfiniBand)
        Net "ib0"
        OpenFabrics "mlx5_0"
  HostBridge
    PCIBridge
      PCI 000d:01:00.0 (InfiniBand)
```

```
17            Net "ib1"
18            OpenFabrics "mlx5_1"
```

Listing 10: Ampere System Topology

A.7 Block ID - Texel Address Conversion

Since OpenMP ensures each thread receives an independent block of loop indices, these indices can then be used to uniquely identify blocks within an image. The function `id_to_index` converts a block ID into the x and y coordinates of the first texel of the block to be extracted (i.e., the top-left pixel of the block). With this index, the `extract_block` function extracts the rest of the texels from the image, accounting for any padding at the end of each row through the use of the stride parameter, which indicates the actual number of bytes per row.

```
/* id:    Block ID
 * w/h:   Image width/height
 * bw/bh: Block width/height,
 * x/y:   Pixel output coordinates */
void id_to_index(int id, uint32_t w, uint8_t bw, uint8_t bh,
                 uint32_t * x, uint32_t * y) {
    *x = id % (w / bw) * bw;
    *y = id / (w / bw) * bh;
}

/* raw:     Input RGB buffer, out: Output buffer,
 * convert: Whether to convert to intermediate values or leave the data as-is
 * stride:  Bytes per line of pixel data, including any padding at the end
 * x/y:     Top-left pixel coordinates
 * bw/bh:   Block width/height */
void extract_block(uint8_t * raw, void * out, uint8_t convert, uint64_t stride,
                   int64_t x, int64_t y, uint8_t bw, uint8_t bh) {
    assert(x % bw == 0);
    assert(y % bh == 0);
    uint64_t pitch = 3; // 24 bit
    uint64_t iw = 0;
    for (uint64_t iy = y; iy < y + bh; iy++) {
        if (convert) {
            for (uint64_t ix = x; ix < x + bw; ix++) {
                uint8_t * ptr = raw + iy * stride + ix * pitch;
                ((ir_texel *)out)[iw].r = (ir_fmt)ptr[0];
                ((ir_texel *)out)[iw].g = (ir_fmt)ptr[1];
                ((ir_texel *)out)[iw].b = (ir_fmt)ptr[2];
                iw++;
            }
        } else {
            uint8_t * ptr = raw + iy * stride + x * pitch;
            memcpy(((uint8_t *)out) + iw, ptr, stride * bw);
            iw += stride * bw;
        }
    }
}
```

Listing 11: Supplementary block extraction functions