Characterizing the Performance of Parallel Data-Compression Algorithms across Compilers and GPUs

Brandon Alexander Burtchell
Texas State University
San Marcos, TX, USA
burtchell@txstate.edu

Martin Burtscher
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

Abstract

Different compilers can generate code with notably different performance characteristics—even on the same system. Today, GPU developers have three popular options for compiling CUDA or HIP code for GPUs. First, CUDA code can be compiled by either NVCC or Clang for NVIDIA GPUs. Alternatively, AMD's recently introduced HIP platform makes porting from CUDA to HIP relatively simple, enabling compilation for AMD and NVIDIA GPUs. This study compares the performance of 107,632 data-compression algorithms when compiling them with different compilers and running them on different GPUs from NVIDIA and AMD. We find that the relative performance of some of these codes changes significantly depending on the compiler and hardware used. For example, Clang tends to produce relatively slow compressors but relatively fast decompressors compared to NVCC and HIPCC.

CCS Concepts

• **Theory of computation → Data compression**.

Keywords

Data compression, GPU compilers, CUDA, HIP

ACM Reference Format:
Brandon Alexander Burtchell and Martin Burtscher. 2025. Characterizing the Performance of Parallel Data-Compression Algorithms across Compilers and GPUs. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767369

1 Introduction

The choice of compiler can affect the performance of a program. Typically, this is due to the optimizations the compiler applies to the code—such as strength reduction or loop unrolling [15]. Different compilers support different sets of optimizations and use different algorithms for important code-generation steps like register allocation. Moreover, they may include different versions of libraries. These issues are exacerbated for GPU code, where the compilers, language features, and libraries are not as mature as for CPUs. Expecting every developer to compare such low-level details between candidate compilers to choose one is unrealistic. Hence, with the growing prominence of GPU workloads, it is important to conduct performance analyses to better understand how the main compilers compare. We use 62 transformations from data-compression algorithms for this purpose.

There are several popular options for compiling general-purpose GPU code. CUDA [7]—NVIDIA's proprietary API—can be compiled by NVIDIA's NVCC compiler or the open-source LLVM-based Clang compiler [5]. AMD's HIP [8]—an open-source API for NVIDIA *and* AMD GPUs—can be compiled by the HIPCC compiler. Existing CUDA code can be easily ported to HIP via the HIPIFY utility [3], making it important to compare the performance between the same codes represented in CUDA and HIP. Other compilers like PGI [9] (now part of the NVIDIA HPC SDK [1]) are less widely used.

The choice of GPU vendor can affect performance due to architectural characteristics. For example, NVIDIA has yet to release a GPU with a warp size above 32 threads, while AMD has released several GPUs (e.g., the Instinct generation of accelerators) with a warp size of 64 threads. Thus, code that is able to exploit larger warp sizes (e.g., warp-based reductions) can achieve more warp-level parallelism on such AMD GPUs than on NVIDIA GPUs. On the other hand, while both HIP and CUDA support system- and device-scope atomics, only CUDA supports block-scope atomics that merely guarantee atomicity between threads from the same block but are, therefore, potentially faster.

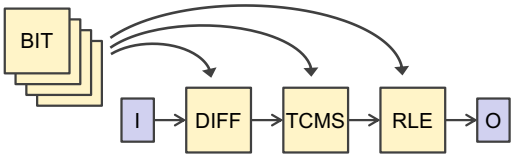


Figure 1: A compression pipeline with 3 stages generated by the LC framework (I = input, O = output)

To understand the effect of the compiler and hardware vendor on program performance, we need a diverse set of codes that can be measured in various contexts. For this study, we use LC [13]—a framework that automatically generates high-speed and effective data-compression algorithms, called *pipelines*, for GPUs. LC has been used to create several state-of-the-art lossless and lossy compressors including SPspeed, SPratio, DPspeed, DPratio, and PFPL [14, 18]. LC synthesizes pipelines by chaining data transformations called *components*. Each component has different algorithmic characteristics, such as the granularity at which it operates, the transformation it performs, and its computational complexity. Fig. 1

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767369

visualizes how a compression pipeline is formed from a library of components. The corresponding decompression pipeline performs the inverse transformations in reverse order.

All components have a common interface such that each of them can be given a block of input data and transform it into a block of output data, which can then be fed into the next component. Thus, the position of a component can affect its runtime since a preceding component determines the size of its input. With 62 components in its library, LC can generate 107,632 three-stage pipelines.

We investigate the performance of the 62 data transformations in the context of these 107,632 unique lossless data-compression algorithms when compiled by NVCC, Clang, and HIPCC. Moreover, we evaluate the speed of the resulting codes on several NVIDIA and AMD GPUs from different generations. Our experiments reveal that the LC components are not individually sensitive to the choice of compiler or GPU vendor, but we find trends between components that may apply to other codes with similar algorithmic patterns.

This work makes the following main contributions.

- It is, as far as we know, the first study to compare CUDA and HIP in the domain of data compression.
- It highlights interesting performance trends between compiled codes across several compilers and GPUs.
- It updates the LC framework for compatibility with the NVCC, Clang, and HIPCC compilers.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 provides an overview of the tested compilers and the LC framework. Section 4 explains how we updated LC for compatibility. Section 5 outlines the experimental methodology. Section 6 presents and analyzes the results. Section 7 provides a summary and conclusions.

2 Related Work

Several works compare different general-purpose GPU programming models. In particular, CUDA and OpenCL [27] have been extensively compared [19, 21]. These works find that CUDA generally yields similar or slightly better performance than OpenCL. Furthermore, Martineau et al. [23] compare OpenMP 4.5's GPU support [26] against CUDA to identify and implement optimizations in OpenMP when compiled with Clang. Our work differs in that we compare the same codes in the same (CUDA) or near-identical (HIP) programming model, compiled by different compilers and tested on various GPUs from NVIDIA and AMD.

Some existing works directly compare NVCC and Clang when compiling CUDA code. Wu et al. [29] introduce Clang's CUDA support under the name `gpucc` and compare Clang with NVCC to find that Clang's compilation is typically faster and runtime performance is on par with NVCC. Balogh et al. [16] compare several languages (CUDA, OpenACC [2], OpenMP), compilers (including NVCC, Clang, and PGI), and parallelization approaches on unstructured mesh computations. They find that Clang frequently outperforms NVCC on CUDA code for their applications. In contrast, our work compares a different set of compilers and programming models and focuses on the domain of data compression.

As HIP is relatively new compared to the platforms mentioned above, there is little existing work that measures HIP codes ported from CUDA. Kondratyuk et al. [22] port molecular dynamics applications from existing CUDA/OpenCL versions to HIP and find that the tested HIP ports generally outperform OpenCL implementations on the same AMD GPU. Tsai et al. [28] port the Ginkgo linear algebra package [10] and find that HIP introduces only negligible overhead (3-10% performance difference) on their benchmarks. To the best of our knowledge, our work is the first to present a CUDA and HIP comparison in the domain of data compression.

Azami and Burtscher [12] analyze the importance of LC components in terms of compression ratio. They find that various stages prefer distinct component types, and that the preferred word size of certain components depends on the data type of the input (i.e., single- vs. double-precision data). This approach has inspired our work, but we analyze the encoding and decoding throughputs of components instead. Additionally, our work adds the dimensions of compiler, GPU, and GPU vendor to the comparison.

3 Background

This section describes the tested GPU compilers as well as the analyzed codes.

3.1 GPU Compilers

General-purpose GPU code targeting NVIDIA platforms is typically written in CUDA, NVIDIA's C++ API. NVIDIA provides NVCC as the de facto compiler for CUDA code. NVCC handles the compilation of device (i.e., GPU) code and forwards all host (i.e., CPU) code to `g++`. Notably, NVCC is a proprietary compiler. This can presumably result in fast code, as the compiler can employ optimizations specific to the proprietary design of the target hardware. However, it can also make performance analysis of GPU codes more difficult since aspects of the hardware and the compiler are not publicly known. Additionally, this means that CUDA code cannot be compiled with NVCC for hardware from other vendors.

Clang is an open-source C/C++ compiler built in tandem with the LLVM compiler backend. Initially introduced under the name `gpucc` [29], CUDA support has been available in Clang as of LLVM version 3.9 [6]. Clang is currently the most popular open-source alternative to NVCC. Like NVCC, Clang can only compile CUDA codes for NVIDIA GPUs.

AMD introduced HIP as part of the Radeon Open Compute platform (ROCm) [4]. HIP is a C++ runtime API and platform that enables the creation of portable applications that target both NVIDIA and AMD GPUs. The API is designed in a way to make porting CUDA to HIP simple. In fact, almost all CUDA API functions have a HIP counterpart, meaning many applications can be ported with a simple find-and-replace. For example, `cudaMemcpy()` becomes `hipMemcpy()`. HIP provides HIPIFY tools to automate this process for developers.

For HIP, both AMD and NVIDIA targets require the HIP/ROCm libraries, but compilation is necessarily handled differently. For AMD GPUs, HIP code is compiled with AMD's HIPCC—an open-source compiler built with LLVM. For NVIDIA GPUs, HIPCC simply invokes NVCC while including the HIP/ROCm libraries. Thus, the only difference between NVCC-compiled CUDA code and NVCC-compiled HIP code is which API libraries are referenced when calling CUDA or HIP functions.

3.2 LC Framework

LC-generated data compressors split an input file into 16 kB chunks and operate on the chunks in parallel by assigning them to different thread blocks. The code processing each chunk is further parallelized using thread, warp, and block-local primitives. The overall activity is synchronized via a global prefix sum.

Table 1: List of LC components by category

Mutators	Shufflers	Predictors	Reducers
DBEFS_j	BIT_i	DIFF_i	CLOG_i
DBESF_j	TUPLk_i	DIFFMS_i	HCLOG_i
TCMS_i		DIFFNB_i	RARE_i
TCNB_i			RAZE_i
			RLE_i
			RRE_i
			RZE_i

Table 1 shows the list of LC components used in this study. Most of them are derived from common data transformations performed in various data compressors. We categorize them into four types: mutators, shufflers, predictors, and reducers. The first three types leverage correlations to better expose data patterns so that the reducers can compress more effectively. The parameters i, j, and k in the table denote the granularity at which these components operate. For instance, TCMS_4 operates on 4-byte words. The supported word sizes for i are 1, 2, 4, and 8 bytes (with the exception of TUPL), and for j they are 4 and 8. The TUPL components assume the input to be a sequence of tuples of various sizes. The k value in the table represents the tuple size. We tested 6 TUPL components with sizes of 2, 4, and 8—each with their own set of word granularities.

The following subsections briefly describe the types of transformations performed by these components. Table 2 summarizes the work complexity and span of each component's encoding and decoding algorithm, where n is the number of words in the input data and w is the word size.

3.2.1 Mutators. Mutators computationally transform each value in place without compressing the data. When decoding, these components perform the inverse transformation.

- **DBEFS**: Operates on IEEE-754 floating-point values. It first de-biases the exponent and then rearranges the data fields from sign, exponent, fraction order to (de-biased) exponent, fraction, sign order.
- **DBESF**: Like DBESFS, but rearranges to (de-biased) exponent, sign, fraction order.
- **TCMS**: Converts each value from two's complement to magnitude-sign representation.
- **TCNB**: Converts each value from two's complement to base negative 2 (a.k.a. negabinary) representation.

3.2.2 Shufflers. Shufflers rearrange the order of values but perform no computation or compression on them. To decode, these components apply the rearrangement in reverse.

- **BIT**: Takes the most significant bit of each value in the input and outputs them together, then it does the same with the second bit and so on. The actual GPU implementations vary slightly with the word size in order to utilize warp-level parallelism at the larger word sizes.
- **TUPL*k***: Assumes the data to be a sequence of k-tuples, which it rearranges by listing all first tuple values, then all second tuple values, etc. For example, a tuple size of $k = 2$ changes the sequence $x_1, y_1, x_2, y_2, \ldots$ into $x_1, x_2, \ldots, y_1, y_2, \ldots$.

3.2.3 Predictors. Predictors guess the next value by extrapolating it from prior values. Then they subtract the prediction from the actual value to output a sequence of residuals. If the predictors are accurate, the residuals cluster around zero, making them easier to compress by a subsequent reducer.

- **DIFF**: Computes the difference sequence (a.k.a. "delta modulation") by subtracting the previous value from the current value and outputting the resulting difference. To decode, the prefix sum of the differences is computed—yielding the original values.
- **DIFFMS** and **DIFFNB**: Like DIFF but stores each value in magnitude-sign format for DIFFMS or negabinary format for DIFFNB.

3.2.4 Reducers. Reducers are the only components that can compress the data. They do so by exploiting redundancies.

- **CLOG**: Breaks each 16 kB chunk into 32 subchunks, determines the smallest number of leading zero bits of all values in a subchunk, records this count, and then stores only the remaining bits of each value. This compresses data with leading zero bits. Decoding uses the recorded leading-zero-bit counts to recreate the values of each subchunk.
- **HCLOG**: Like CLOG except it first applies the TCMS transformation to all values in a subchunk that yields no leading zero bits when using CLOG.
- **RLE**: Performs run-length encoding. It counts how many times a value appears in a row, then it counts how many non-repeating values follow. Both counts are emitted followed by a single instance of the repeating value as well as all non-repeating values. To decode, the values are emitted according to the encoded runs.
- **RRE**: Creates a bitmap where each bit specifies whether the corresponding word in the input repeats the prior word. It outputs the non-repeating words and a compressed version of the bitmap that is repeatedly compressed with the same algorithm. To decode, the bitmap is decompressed, then the values are emitted accordingly.
- **RZE**: Like RRE, but the bitmap specifies whether the corresponding word is a zero or not.
- **RARE**: Treats the upper k bits separately from the lower bits by only applying RRE to the upper k bits and always keeping the lower bits. It automatically picks the optimal k value for each chunk. Decoding similarly performs the RRE decoding procedure on only the encoded k bits.
- **RAZE**: Like RARE, except it applies the RZE transformation to the upper k bits.

4 Approach

To make our experiments possible, we first had to make LC's CUDA implementation compatible with Clang and HIPCC as well as AMD

Table 2: Component work complexity and span in big-O notation

		Mutators		Shufflers		Predictors	Reducers			
		DBEFS/ESF	TCMS/NB	BIT	TUPL	DIFF, DIFFMS/NB	CLOG, HCLOG	RARE/ZE	RLE	RRE/ZE
Enc.	Work	n	n	$n \log w$	n	n	n	n	n	n
	Span	1	1	$\log w$	1	1	1	$\log n$	$\log n$	$\log n$
Dec.	Work	n	n	$n \log w$	n	n	n	n	n	n
	Span	1	1	$\log w$	1	$\log n$	1	$\log n$	1	$\log n$

GPUs. Luckily, LC compiled successfully for NVIDIA GPUs by Clang without any modifications. However, after initially converting the CUDA code to HIP using HIPIFY, we needed to update parts of LC that could not be automatically ported. For instance, as of HIP 6.2, some CUDA primitives like the block-scope versions of atomic functions (e.g., `atomicOr_block()`) are not supported. In this particular case, it is possible to fall back to the device-scope version (e.g., `atomicOr()`) and maintain correctness at the possible cost of some performance. We applied similar workarounds for `atomicAdd_block()`, `__syncwarp()`, and `__trap()`.

```
1  int tmp, val = ...;
2  const int lane = threadIdx.x % WS;
3
4  tmp = __shfl_up(val, 1);
5  if (lane >= 1) val += tmp;
6  tmp = __shfl_up(val, 2);
7  if (lane >= 2) val += tmp;
8  tmp = __shfl_up(val, 4);
9  if (lane >= 4) val += tmp;
10 tmp = __shfl_up(val, 8);
11 if (lane >= 8) val += tmp;
12 tmp = __shfl_up(val, 16);
13 if (lane >= 16) val += tmp;
14
15 #if defined(WS) && (WS == 64)
16 tmp = __shfl_up(val, 32);
17 if (lane >= 32) val += tmp;
18 #endif
```

Listing 1: CUDA/HIP prefix-sum code updated to support both 32 and 64 threads per warp

Most notably, we had to update several parts of the code that were written with the assumption that the warp size is 32 threads. For example, Listing 1 shows a snippet of CUDA/HIP code that performs a prefix sum across a warp. Each thread starts with its own value in `val`, then each `__shfl_up(int var, int delta)`[1] call returns the private value stored in `var` from the lane whose ID is equal to the current lane ID minus `delta`. That is, `__shfl_up(val, 2)` returns the value 2 lanes below the calling lane. The code before Line 15 only supports warps with 32 threads. We added a preprocessor directive and extra code after Line 15 that is only included during compilation if the warp size (`WS`) is 64 threads. Several components such as CLOG, HCLOG, RARE, RAZE, and RLE perform similar warp-level operations that we updated to work on AMD GPUs with 64-thread warps.

5 Experimental Methodology

To analyze the performance of each individual component, we created 3-stage LC pipelines from the components listed in Table 1. Since placing a non-reducer in the final stage is useless, we limit the last stage to reducers only. Thus, we end up with $62 \times 62 \times 28 = 107{,}632$ generated pipelines.

We record the encoding and decoding throughputs (uncompressed bytes processed per second) for all pipelines on all inputs—repeating this process for each tested compiler on each tested GPU. To analyze the resulting throughputs in terms of individual components, we separately average the throughputs across several dimensions, namely: GPU, component word size, component type, and component.

Table 3: SP dataset

Filename	Size (MB)
msg_bt	133.2
msg_lu	97.1
msg_sp	145.1
msg_sppm	139.5
msg_sweep3d	62.9
num_brain	70.9
num_comet	53.7
num_control	79.8
num_plasma	17.5
obs_error	31.1
obs_info	9.5
obs_spitzer	99.1
obs_temp	20.0

We used the SP dataset [17] as inputs, which are summarized in Table 3. This dataset includes 13 single-precision floating-point files from several domains (e.g., weather observations, simulation results, MPI messages). For each input, we run each LC pipeline three times and compute the throughput based on the median runtime. Then, each pipeline's throughput is averaged across all 13 inputs using the geometric mean.

Table 4: NVIDIA GPU specifications

	TITAN V	3080 Ti	4090
Clock Freq. (MHz)	1075	1755	2625
SMs	24	80	128
Max Threads per SM	2048	1536	1536
Warp Size	32	32	32
Memory (GB)	12	12	24
Compute Capability	7.0	8.6	8.9

We perform our experiments on the 3 NVIDIA GPUs listed in Table 4 and the 2 AMD GPUs listed in Table 5. Some architectural items differ between vendors. Namely, NVIDIA's streaming multiprocessors (SMs) ≈ AMD's compute units (CUs) and NVIDIA's

[1] This is `__shfl_up_sync(int mask, int var, int delta)` in CUDA.

Table 5: AMD GPU specifications

	MI100	RX 7900 XTX
Clock Freq. (MHz)	1502	2482
CUs	120	96
Max Threads per CU	2560	1024
Warp Size	64	32
Memory (GB)	32	24
Target Processor	`gfx908`	`gfx1100`

compute capability ≈ AMD's target processor. Note that the AMD RX 7900 XTX is based on the RDNA3 architecture, which combines pairs of CUs into *workgroup processors* in which all threads can cooperate.

The system with the TITAN V and RTX 4090 contains an AMD Ryzen Threadripper 2950X CPU with 48 GB of memory. The system with the RTX 3080 Ti contains an Intel Xeon Gold 6226R CPU with 64 GB of memory. Both systems run Fedora 41 and NVIDIA GPU driver 550.142. The system with the MI100 contains an AMD EPYC 7V13 CPU with 512 GB of memory—running Ubuntu 20.04 and AMD GPU driver 5.16.9.22.20. The system with the RX 7900 XTX contains an AMD Ryzen Threadripper 3970X CPU with 256 GB of memory—running Ubuntu 24.04 and AMD GPU driver 6.8.5.

Note that each tested input fully occupies all tested GPUs due to how LC assigns a 16 kB chunk of data to each 512-thread block. For instance, the RTX 4090 has 128 SMs with 1536 threads per SM (i.e., 3 blocks per SM). Therefore, it takes 6 MB of input data to fully occupy this GPU. Similarly, it takes 9.375 MB to fully occupy the AMD MI100. These are the tested GPUs with the most active threads for both vendors, meaning all tested inputs (the smallest being "obs_info" at 9.5 MB) fully occupy the tested GPUs. Larger files take up more global memory, but the presented throughputs normalize the effect of input size.

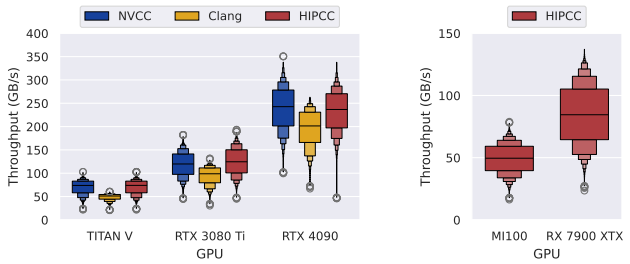
All NVIDIA systems use NVCC version 12.6, HIPCC version 6.2, and Clang 18.1. For the AMD systems, the system with the MI100 uses HIP version 6.4 and the system with the RX 7900 XTX uses HIP version 6.2. All compilers target C++17. We compiled separate executables for each GPU's compute capability/target processor. All results except those in Section 6.5 present throughputs from executables compiled with the `-O3` optimization flag. Section 6.5 presents speedups over using the `-O1` flag.

6 Results

The following subsections present the throughputs across compilers when using various GPUs, component word sizes, component types, components, and optimization levels. To show the distribution of throughputs across all pipelines within a particular parameter, we use boxen plots (a.k.a. letter-value plots) [20], which recursively halve the distribution around the median and present each quantile. That is, the widest box represents the middle 50% of values and the next two narrower boxes above and below together contain the next 25% of values and so on. The median is marked within the widest box as a black line. Outliers are set to a fixed rate of 0.7% and are shown as gray circles. The figures in Sections 6.1 through 6.4 present throughput (higher is better) on the y-axis in terms of GB/s, with boxes grouped by the parameter of interest along the x-axis. Each box is color-coded according to the compiler used. With the exception of the figures in Sections 6.1 and 6.5, all figures show results from the fastest tested NVIDIA/AMD GPU. We observed similar performance patterns across all GPUs—relative to each GPU's compute power. Since each platform supports a different set of compilers, we separate all subsequent figures into a subfigure for NVIDIA and another for AMD. The purpose of this work is not to compare the performance of the tested GPUs but to characterize relative performance trends across several parameters. Therefore, the NVIDIA and AMD subfigures use different y-axis scales.

6.1 GPU Comparison

Fig. 2 summarizes the encoding throughputs of all pipelines for each compiler on each tested GPU. For both vendors, the more powerful and newer GPUs naturally have higher overall performance, hence the staircase shape in each subgraph (i.e., TITAN V to 4090 and M100 to 7900 XTX). Overall, each encoding throughput distribution is quite symmetric. That is, the "head" and "tail" of each box have similar lengths and the median is well-centered.



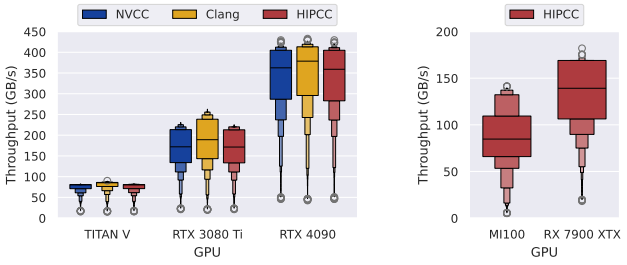
(a) NVIDIA **(b) AMD**

Figure 2: Encoding throughputs by GPU

There are some interesting trends between the compilers on the NVIDIA platform (Fig. 2a). NVCC and HIPCC's distributions are always close. Recall that the only difference between NVCC and HIPCC targeting NVIDIA GPUs is the included libraries. These results suggest that the functions LC uses from the CUDA and HIP libraries have similar performance when compiled for NVIDIA GPUs. In contrast, Clang's encoding throughputs are consistently lower than those from NVCC and HIPCC.

Fig. 3 shows the decoding throughputs, which are generally higher than the corresponding encoding throughputs. This is expected since the encoding version of each component tends to do more work than its decoding counterpart. We observe this for each compiler, GPU, and vendor.

Interestingly, the shape of the distributions also differs between encoding and decoding. Specifically, the decoding throughput distributions are not symmetric but skew towards higher throughputs, indicating that decoding is consistently very fast for the majority of LC pipelines. Regarding the compilers for NVIDIA GPUs, NVCC and HIPCC are once again close in behavior like they were for encoding. However, unlike the encoding throughputs, Clang's decoding throughputs are consistently *higher* than those of NVCC



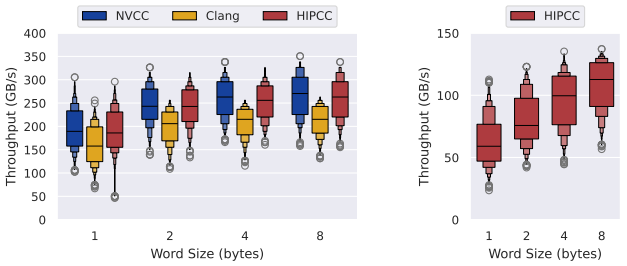
(a) NVIDIA (b) AMD

Figure 3: Decoding throughputs by GPU

and HIPCC. This seems to be a general trend that holds for other parameters as discussed below. Therefore, the performance difference must stem from aspects of the LC framework that are shared among all pipelines yet only occur in the encoder or decoder. For instance, the encoder uses Merrill and Garland's decoupled look-back technique [24] to propagate the cumulative size of prior compressed chunks to the next thread block so it knows where to write its output. In contrast, the decoder computes a prefix sum in each thread block to determine the chunk starting positions. It appears that pipeline-independent framework-level operations like these are where Clang generates consistently faster/slower code than NVCC and HIPCC. This is further explored in Section 6.5.

6.2 Word-Size Comparison

This subsection describes the performance characteristics of LC pipelines where all constituent components are of the same word size. For example, "BIT_4 DIFF_4 RRE_4" would be counted among the pipelines with 4-byte components. Mixed word-size pipelines are omitted from these results to highlight the effect of the word size. This yields 1,792 1-byte pipelines, 1,575 2-byte pipelines, 1,792 4-byte pipelines, and 1,575 8-byte pipelines. Fig. 4 shows the results for encoding on the fastest tested NVIDIA and AMD GPUs.

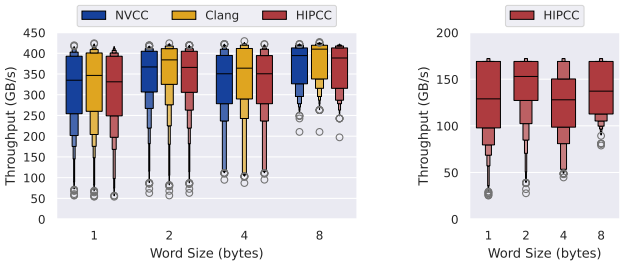


(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 4: Encoding throughputs by wordsize

Encoding throughput generally increases with the word size. Interestingly, despite the fact that all tested GPUs have 32-bit architectures, the 4-byte components do not deliver the highest throughputs. Since the 8-byte components only process half as many words as the 4-byte components on a given input, they still achieve a performance benefit. However, the throughput increase from 4- to 8-byte components is lower than from 2- to 4-byte components due to the architectural word size.

Fig. 5 compares the decoding throughputs for different word sizes. The throughput distributions are close (i.e., the medians are within 50 GB/s of each other and have similarly long tails) for word sizes of 1, 2, and 4 bytes, whereas they trend highest for 8 bytes. This is evident on the NVIDIA and AMD GPUs.



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 5: Decoding throughputs by wordsize

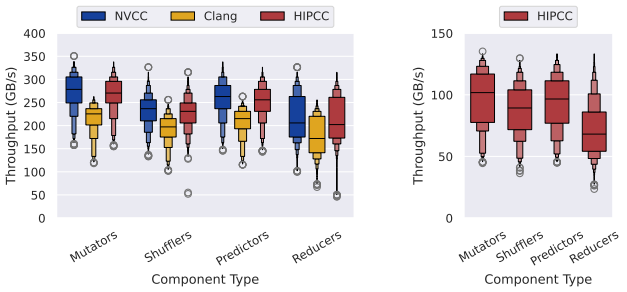
Again, we see that the pipelines with 4-byte word sizes are not necessarily the fastest despite matching the architectural word size. Furthermore, for both vendors and all compilers, the median throughputs are higher for a word size of 2 bytes than for 1 or 4 bytes. This is most evident in Fig. 5b, where the 2-byte word size achieves the highest median decoding throughput on the AMD GPU. This is a side effect of a phenomenon explored in Section 6.4, where reducers (notably RLE) in Stage 1 that do not match the word size of the data (i.e., 4-byte floats) fail to compress the data and thus require little work during decoding. In contrast, a reducer that matches the input's word size tends to compress and, therefore, must decompress during decoding—resulting in lower decoding throughput.

For both encoding and decoding, we see that the choice of compiler does not discriminate between word sizes. All compilers exhibit the same relative trends.

6.3 Component-Type Comparison

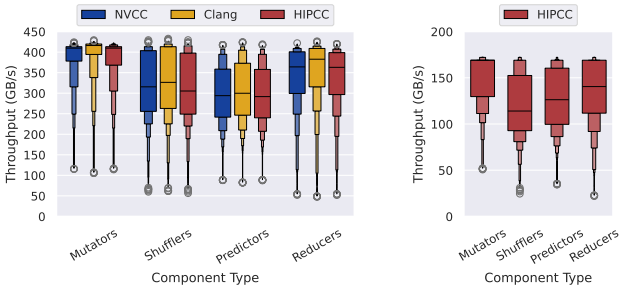
This subsection presents the performance characteristics of 3-stage pipelines where the first two stages contain components that are of the same type. For example, "TCMS_4 DBEFS_1 HCLOG_2" would be counted among the pipelines with mutators in the first two stages. Recall that the last stage must always be a reducer. Accordingly, there are 4,032 mutator pipelines, 2,800 shuffler pipelines, 4,032 predictor pipelines, and 21,952 reducer pipelines that we evaluated.

Fig. 6 shows the encoding throughputs. Here, the component types yield similar throughputs except for the reducers. This is expected since reducers tend to perform the most complex transformations and, therefore, more work during encoding than the other components. They typically also necessitate more synchronization in their parallel implementations.



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 6: Encoding throughputs by component type



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 7: Decoding throughputs by component type

Fig. 7 presents the corresponding decoding throughputs, where the reducers are no longer the slowest. Rather, the pipelines with predictors tend to yield the lowest throughputs because they require expensive prefix sums. Interestingly, the throughput distribution of the pipelines with mutators in the first two stages is highly skewed towards the top—so much so that the median lines are barely visible. This is due to the mutators' embarrassingly parallel implementations in combination with regular memory accesses. In contrast, the shufflers exhibit irregular memory accesses, and the predictors and reducers perform complex operations that are not embarrassingly parallel.

Once again, the choice of compiler does not discriminate between component type. Moreover, we again observe the overall trends on NVIDIA GPUs where NVCC and HIPCC tend to match performance and Clang is consistently slower for encoding but faster for decoding.

6.4 Component Comparison

The following figures show results for pinning a particular component to a particular pipeline stage. For example, in Fig. 8, the group of boxes for BIT refers to all pipelines where BIT_1, BIT_2, BIT_4, or BIT_8 are in Stage 1. The other stages can be any other component, with the last stage limited to reducers only. With these constraints, for Stage 1, there are 6,944 pipelines for each component except DBEFS and DBESF, which each have 3,472 pipelines, and TUPL, which has 10,416 pipelines. The components are alphabetically ordered along the x-axis.

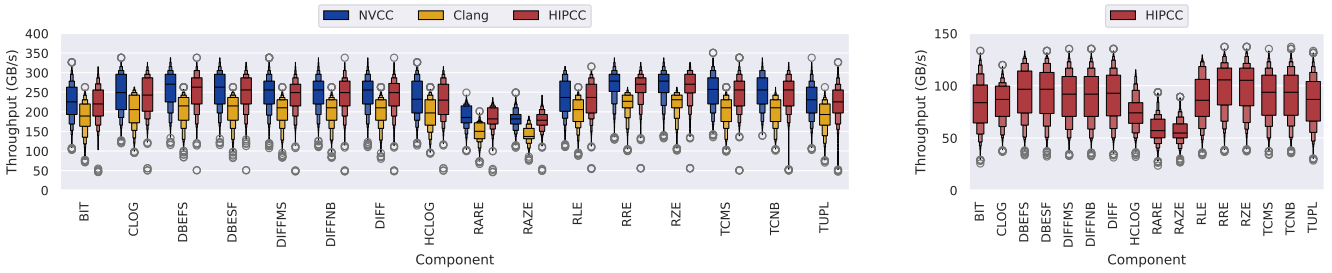
Fig. 8 shows encoding throughputs for pipelines with a given component pinned to Stage 1. Like with the previous figures, the encoding throughput distributions are symmetric. Furthermore, they are close for many of the components. However, on all GPUs, pipelines with RARE and RAZE in the first stage have significantly lower throughputs than the other components—due to their adaptive algorithm performing more work than the other reducers. Additionally, the HCLOG components also have markedly lower throughputs due to their relatively complex operation, especially on the 7900 XTX. On the MI100, our other AMD GPU, the HCLOG behavior is closer to that on the NVIDIA GPUs (results not shown).

Fig. 9 shows the corresponding decoding throughputs when a component is pinned to Stage 1. We observe that CLOG, HCLOG, RRE, and RZE (reducers) tend to have the highest median throughputs. Like the decoding throughputs in previous subsections, almost all of the distributions trend upwards. However, when grouped by component, we see that some components do not follow this pattern. Namely, the decoding versions of BIT and RLE have a wide spread of throughputs for the middle 50% (largest box). The median throughput is also more centered than for the other components. This shows that pipelines with RLE or BIT components in the first stage vary more significantly in overall throughput than pipelines with another component in the first stage. To verify this, Figs. 10 and 11 present detailed views of the Stage 1 results for each word size of the BIT and RLE components, respectively.

Indeed, the decoding throughputs vary significantly between word sizes for BIT and RLE. For the BIT components (Fig. 10), the BIT_1 and BIT_2 distributions skew towards higher throughputs (more so for BIT_2) much like the other decoding components, whereas the BIT_4 and BIT_8 distributions are symmetric. This is primarily due to the very different implementations of BIT_1/BIT_2 on the one hand and BIT_4/BIT_8 on the other hand. In particular, BIT_1 and BIT_2 perform bitwise operations without synchronization whereas BIT_4 and BIT_8 perform `__shfl_xor()` operations (which implicitly synchronize across warps). We also observe these trends for encoding pipelines with BIT in Stage 1, though with less severity and more symmetric distributions.

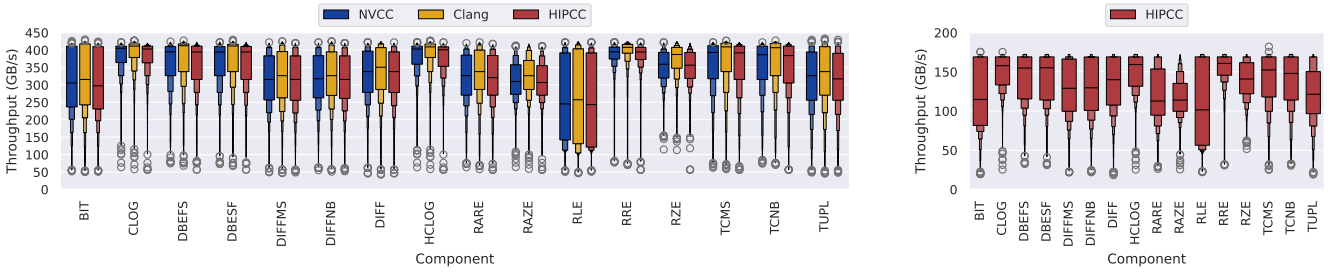
The decoding throughputs for pipelines with an RLE component in the first stage (Fig. 11) differ even more. Unlike with BIT, the RLE components have the same implementation for each word size, so the reason for these discrepancies is different. During LC encoding, if a component *expands* a chunk of data, LC will ignore the expanded output and just copy the original input to the output. Thus, during decoding, LC will not have to run the decoder of that component on that chunk—saving work. Recall that our dataset consists of single-precision (i.e., 4-byte float) inputs. Therefore, it is much more likely for an RLE component in Stage 1 to encounter a "run" of matching 4-byte values (necessary for RLE compression) than a run of matching values at any of the other word sizes. Thus, pipelines with RLE_1, RLE_2, or RLE_8 in Stage 1 tend to have high decoding throughputs because, for these inputs, they typically do not have to perform their decoding procedure. In contrast, RLE_4 tends to be slower because it must perform work to decompress.

We omit the Stage 2 results since the trends echo Stage 1 with the following minor exceptions. Mainly, we see that the decoding



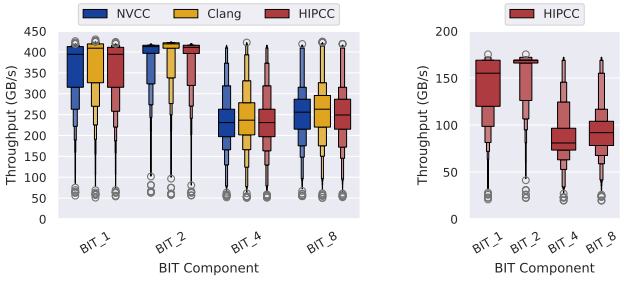
(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 8: Encoding throughputs by component in Stage 1



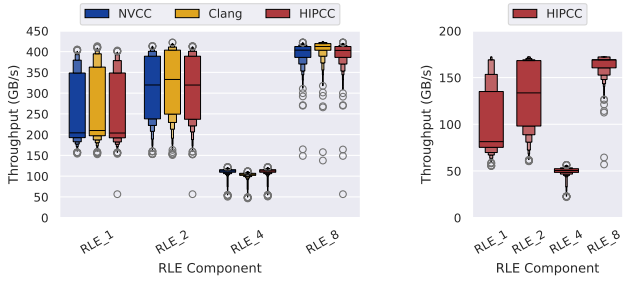
(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 9: Decoding throughputs by component in Stage 1



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 10: Decoding throughputs of pipelines with a BIT component in Stage 1



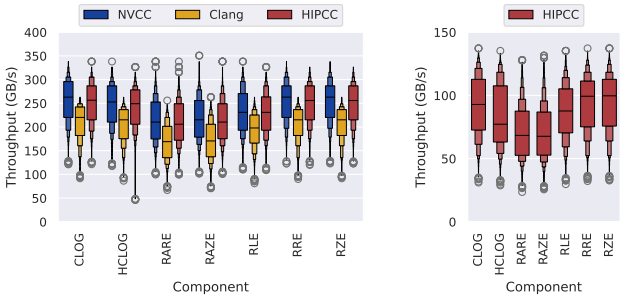
(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 11: Decoding throughputs of pipelines with an RLE component in Stage 1

distributions are more uniform for all components than they are for Stage 1. In particular, RLE no longer exhibits the wide 50% box like it did in Stage 1. Accordingly, pipelines with RLE in Stage 2 have a higher median throughput than pipelines with RLE pinned to Stage 1 (approximately a 100 GB/s improvement). This is because RLE components in Stage 2 receive transformed data from the preceding component in Stage 1. This transformed data is more likely to be similarly compressible by RLE components of different word sizes—alleviating the decoding throughput discrepancies.

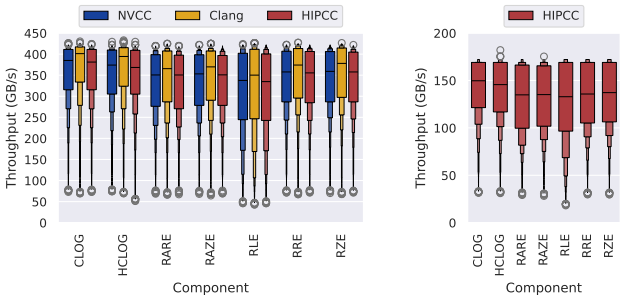
Fig. 12 shows encoding throughputs for pipelines with a component pinned to Stage 3. Recall that Stage 3 can only contain a reducer component, so each distribution represents 15,376 pipelines. Like with stages 1 and 2, RAZE and RARE are the slowest encoding components. HCLOG is again relatively slower on the AMD GPU than on the NVIDIA GPU.

Lastly, Fig. 13 shows decoding results for pinning a component to Stage 3. Overall, we see similar throughput distributions as with the previous stages. Specifically, decoding pipelines with RLE in Stage 3 have the widest distribution of throughputs like they did in



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 12: Encoding throughputs by component in Stage 3



(a) NVIDIA RTX 4090 (b) AMD RX 7900 XTX

Figure 13: Decoding throughputs by component in Stage 3

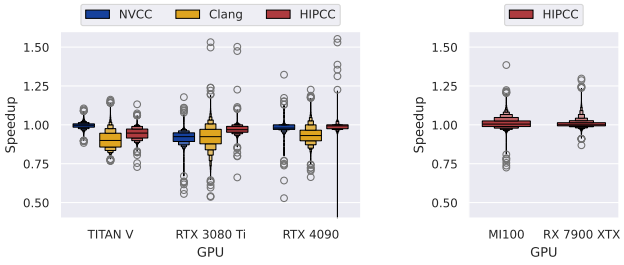
stages 1 and 2. Comparing encoding and decoding, there is again more variability in throughputs for encoding than decoding.

These per-stage results largely reflect the trends seen in the component type analysis in Section 6.3. However, these more fine-grained analyses reveal trends within each component type: mainly that the reducers show the most variability in throughputs across encoding and decoding. Overall, the results maintain that the effect of the chosen compiler is not discriminatory for the various characteristics of the components in the LC framework under a variety of pipeline conditions.

6.5 Optimization-Level Comparison

The previous subsections present results from executables compiled with the `-O3` optimization flag. To investigate the source of the performance discrepancies between Clang and the other compilers, we also evaluated the effect of the compiler optimization flag on overall performance. Fig. 14 shows the distribution of each pipeline's encoding speedups from `-O1` to `-O3` by GPU. Speedups higher than 1.0 mean that `-O3` yields a performance increase.

Most pipelines exhibit negligible speedups/slowdowns between optimization flags. The performance on the AMD GPUs in particular is quite stable. However, on all NVIDIA GPUs, Clang's encoding throughput consistently tends to decrease when going from `-O1` to `-O3`. Clearly, for these codes, the extra `-O3` compiler optimizations in Clang hurt performance for most encoders. Still, the full `-O1`

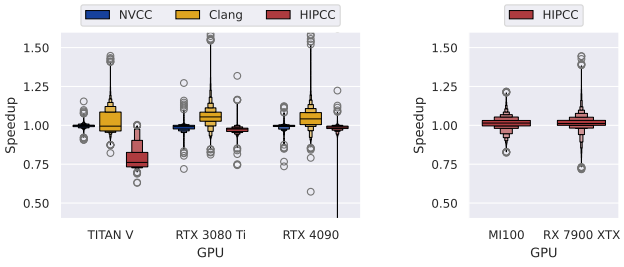


(a) NVIDIA (b) AMD

Figure 14: Encoding speedups from -O1 to -O3 by GPU

results (not shown) maintain that Clang is consistently slower for encoding than the other compilers, suggesting that the choice of optimization flag is not the sole cause for the performance disparity.

Fig. 15 shows the corresponding decoding speedups. Once again, HIPCC's optimizations tend to have a negligible effect on AMD GPUs. The same is true for NVCC and HIPCC on NVIDIA GPUs. However, Clang's decoding performance noticeably improves from `-O1` to `-O3`, but the speedup (less than 10%) does not fully account for the performance discrepancies observed in previous subsections.



(a) NVIDIA (b) AMD

Figure 15: Decoding speedups from -O1 to -O3 by GPU

We conclude that the performance differences between Clang and the other compilers is not only due to compiler optimizations. Other underlying compiler attributes like the register allocation algorithm must be contributing.

7 Summary and Conclusions

This work investigates the performance differences when the same GPU codes are compiled by different compilers for different target GPUs. We use the LC framework, with a library of 62 algorithmic components, to synthesize 107,632 unique three-stage lossless compression pipelines for performance evaluation. Through these experiments, we arrive at a number of observations regarding the tested GPU compilers and the measured algorithmic components. First, there is a negligible performance difference between codes compiled by NVCC or HIPCC for NVIDIA GPUs, suggesting that NVCC compiles the CUDA and HIP library functions into similarly-performing code. In contrast, relative to the other compilers, Clang

is consistently slower for encoding and consistently faster for decoding. Regarding GPU vendors, there are few major discrepancies in relative performance between NVIDIA and AMD GPUs. Furthermore, GPUs from the same vendor demonstrate analogous performance characteristics across generations (relative to their compute power). Overall, the choice of compiler and GPU does not discriminate between the word size, component type, or algorithm of an LC component. Nonetheless, there are several interesting performance trends. For example, a larger word size tends to lead to better performance (except for BIT and RLE components). Generally, reducers show the most variability in throughput for encoding and decoding due to their data-dependent behavior and compressibility.

Although the generated pipelines are made of components sharing algorithmic steps with other compressors, our findings may only apply to these LC algorithms. Thus, other compressors may be worth studying as well. We expect the relative findings to hold for emerging technologies like NUMA-aware multi-socket GPUs [25] or multi-chip GPUs [11] as long as the shared-memory size of each SM does not significantly decrease, which is unlikely as it has been increasing in recent GPU generations. This is because LC loads entire chunks of data into shared memory before performing any computation. Since this load is performed only once, NUMA latencies would not incur a significant penalty.

We recommend LC users be mindful of the component word size and its relation to the word size of the input. In performance-critical contexts, it may be worth compiling the encoder using NVCC or HIPCC and the decoder using Clang to maximize throughput since LC maintains correctness independently of the used compiler and GPU vendor. For codes in any domain, we recommend developers try both NVCC and Clang for their CUDA applications since doing so requires little to no code changes and can substantially affect performance. Further, we recommend porting to HIP to enable NVIDIA/AMD compatibility since this is relatively easy and tends to yield nearly identical performance to CUDA compiled by NVCC on NVIDIA GPUs and close relative performance on AMD GPUs.

Acknowledgments

This work has been supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Research (ASCR), under contract DE-SC0022223.

References

[1] [n. d.]. NVIDIA HPC SDK. Retrieved August 22, 2025 from https://developer.nvidia.com/hpc-sdk/

[2] [n. d.]. OpenACC. Retrieved August 22, 2025 from https://www.openacc.org/

[3] [n. d.]. ROCm/HIPIFY. Retrieved August 22, 2025 from https://github.com/ROCm/HIPIFY

[4] [n. d.]. AMD ROCm documentation — ROCm Documentation. Retrieved August 22, 2025 from https://rocm.docs.amd.com/en/latest/

[5] [n. d.]. Clang C Language Family Frontend for LLVM. Retrieved August 22, 2025 from https://clang.llvm.org/

[6] [n. d.]. Compiling CUDA with clang. Retrieved August 22, 2025 from https://llvm.org/docs/CompileCudaWithLLVM.html

[7] [n. d.]. CUDA C++ Programming Guide. Retrieved August 22, 2025 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[8] [n. d.]. HIP Documentation. Retrieved August 22, 2025 from https://rocm.docs.amd.com/projects/HIP/en/latest/index.html

[9] [n. d.]. PGI Compiler User's Guide. Retrieved August 22, 2025 from https://docs.nvidia.com/hpc-sdk/pgi-compilers/19.10/x86/pgi-user-guide/index.htm

[10] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. 2022. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Trans. Math. Softw.* 48, 1 (Feb. 2022), 2:1–2:33. doi:10.1145/3480935

[11] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto ON Canada, 320–332. doi:10.1145/3079856.3080231

[12] Noushin Azami and Martin Burtscher. 2025. Identifying Important Data Transformations for Synthesizing Effective Lossless Compressors. In *Proceedings of the 2025 IEEE International Symposium on Performance Analysis of Systems and Software*. doi:10.1109/ISPASS64960.2025.00034

[13] Noushin Azami, Alex Fallin, Brandon Burtchell, Andrew Rodriguez, Benila Jerald, Yiqian Liu, Anju Mongandampulath Akathoott, and Martin Burtscher. [n. d.]. LC Git Repository. https://github.com/burtscher/LC-framework

[14] Noushin Azami, Alex Fallin, and Martin Burtscher. 2025. Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 395–409. doi:10.1145/3669940.3707280

[15] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. doi:10.1145/197405.197406

[16] G. D. Balogh, I. Z. Reguly, and G. R. Mudalige. 2018. Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Springer, Cham, 22–43. doi:10.1007/978-3-319-72971-8_2

[17] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *2007 Data Compression Conference (DCC'07)*. IEEE, Snowbird, UT, USA, 293–302. doi:10.1109/DCC.2007.44

[18] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. 2025. Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium*. doi:10.1109/IPDPS64566.2025.00083

[19] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A Comprehensive Performance Comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*. 216–225. doi:10.1109/ICPP.2011.45

[20] Heike Hofmann, Wickham , Hadley, , and Karen Kafadar. 2017. Letter-Value Plots: Boxplots for Large Data. *Journal of Computational and Graphical Statistics* 26, 3 (July 2017), 469–477. doi:10.1080/10618600.2017.1305277

[21] Kamran Karimi, Neil G. Dickson, and Firas Hamze. 2011. A Performance Comparison of CUDA and OpenCL. doi:10.48550/arXiv.1005.2581

[22] Nikolay Kondratyuk, Vsevolod Nikolskiy, Daniil Pavlov, and Vladimir Stegailov. 2021. GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP. *The International Journal of High Performance Computing Applications* 35, 4 (July 2021), 312–324. doi:10.1177/10943420211008288

[23] Matt Martineau, Simon McIntosh-Smith, Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao, Alexandre Eichenberger, Gheorghe-Teodor Bercea, Tong Chen, Tian Jin, Kevin O'Brien, Georgios Rokos, Hyojin Sung, and Zehra Sura. 2016. Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 54–64. doi:10.1109/PMBS.2016.011

[24] Duane Merrill and Michael Garland. 2016. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Technical Report NVR-2016-002. NVIDIA. https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

[25] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Cambridge Massachusetts, 123–135. doi:10.1145/3123939.3124534

[26] OpenMP Architecture Review Board. 2015. OpenMP 4.5 Specification. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[27] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (May 2010), 66–73. doi:10.1109/MCSE.2010.69

[28] Yuhsiang M. Tsai, Terry Cojean, Tobias Ribizel, and Hartwig Anzt. 2021. Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP. In *Euro-Par 2020: Parallel Processing Workshops*. Springer, Cham, 109–121. doi:10.1007/978-3-030-71593-9_9

[29] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. gpucc: an open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/2854038.2854041