The More You Buy, The More You Save: A GPU Approach to the 1 Billion Row Challenge

Amol Shah
amols2@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Dev Singh
dsingh14@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Yanni Zhuang
yanniz3@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA

Figure 1. A visual representation of the inspiration for this project. Source: Gemini Nano Banana Pro

1 Introduction

The One Billion Row Challenge (1BRC) is a programming competition that challenges developers to process a large file as quickly as possible[6]. The goal is to read a 13 GB text file containing one billion rows of data in the form "station name;temperature" and calculate the minimum, average, and maximum temperature for each unique station.

While the challenge has been solved extensively across various CPU-centric languages, there has been limited effort to adapt this workload for the GPU. The overhead of transferring data from the CPU to the GPU has overwhelmed the possible performance benefits, resulting in slow runtimes compared to CPU implementations. In standard pipelines, the CPU often becomes the bottleneck, unable to feed the GPU fast enough to saturate its massive parallel compute capabilities. Additionally, text parsing is considered ill-suited for GPUs due to the divergence inherent in variable-length string processing.

This paper proposes a novel GPU implementation of the 1BRC to avoid these bottlenecks by utilizing Microsoft's DirectStorage API [13] in conjunction with NVIDIA's CUDA[8] parallel computing platform. By utilizing DirectStorage, we bypass the CPU's I/O path entirely, enabling high-throughput streaming directly from the NVMe SSD to GPU memory. Once in VRAM, we use custom CUDA kernels for efficient data processing. We demonstrate that by offloading both I/O and computation to the GPU, we can achieve processing speeds that rival those of state-of-the-art CPU implementations.

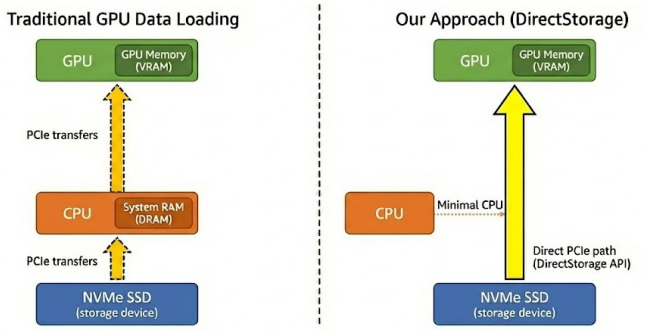


Figure 2. Comparison of traditional GPU data loading vs. our DirectStorage approach.

2 Existing Work

CPU Implementations As part of the original One Billion Row Challenge [6], numerous CPU implementations in Java successfully process the entire dataset. Wuerthinger et al. presents the fastest conventional implementation with a runtime of 1.535 seconds on 8 cores of an AMD EPYC 7502P processor [12]. These implementations leverage aggressive optimizations including memory-mapped I/O, custom hash tables, SWAR (SIMD Within A Register) techniques for parsing, and careful cache-aware data structures. The challenge

establishes strong baselines against which GPU approaches can be evaluated.

GPU-Accelerated Dataframe Libraries Libraries such as NVIDIA's cuDF [10] and its underlying libcudf [11] provide GPU-accelerated DataFrame operations that can technically perform the aggregations required by the 1BRC. cuDF is part of the RAPIDS suite of GPU-accelerated data science libraries and provides a Pandas-like API backed by CUDA C++ implementations. However, these libraries are unsuited for this specific task, as they are designed to support a broad range of DataFrame operations rather than optimized for a single aggregation workload. More critically, these libraries suffer from substantial memory overhead. In our benchmarking, cuDF encountered out-of-memory errors when attempting to process the full 13GB dataset. This limitation may stem from the intermediate data structures and memory management strategies employed by general-purpose DataFrame libraries, which prioritize flexibility over memory efficiency for specific workloads.

Custom CUDA Implementation Kim [4] presents a hand-written CUDA implementation that reports a runtime of 16.8 seconds on an NVIDIA V100 GPU. This work partitions the input file into one million segments, with each CUDA thread processing one segment. The implementation employs atomic operations for concurrent statistic updates and uses binary search over a sorted city list for name lookups. However, several methodological concerns limit the evaluability of this work. First, the implementation deviates from the original challenge specification by requiring prior knowledge of all city names, which are loaded from a separate file and sorted on the CPU before kernel execution. Second, we were unable to compile and reproduce the published artifact. Finally, the author acknowledges several optimization opportunities that remain unexplored, including the use of shared memory for privatization to reduce atomic operation contention, suggesting the reported performance does not represent a well-optimized GPU solution.

3 Technical Approach

Our implementation addresses the fundamental bottleneck in GPU-accelerated data processing: the data transfer overhead between storage, CPU, and GPU. Traditional GPU data copies suffer from a two-stage latency penalty where data must first traverse from NVMe storage to CPU memory, then copy again over PCIe to GPU memory. Traditionally, this penalty is justified by doing some complex computation on the GPU, with the latency reductions from computation overshadowing the transfer time. For a large dataset with relatively simple computations, this PCIe transfer alone can consume many seconds before any computation begins. We bypass the CPU entirely by leveraging the DirectStorage API, enabling direct memory access from NVMe storage to GPU memory.

Our code employs a pipelined architecture with three main components working in concert. First, 16 DirectStorage queues operate in round-robin fashion to maximize I/O parallelism and saturate the NVMe bandwidth as much as possible. Second, 32 pre-allocated stream buffers serve as the data staging area, with each buffer holding approximately 32 MB of data, including a 64 KB overlap region to handle records spanning chunk boundaries. Third, CUDA streams provide asynchronous compute execution, allowing I/O and processing to overlap temporally. This design ensures that while some buffers are being filled from storage, others are being processed on the GPU, and yet others are being cleaned up and prepared for reuse.

We configured the DirectStorage runtime to manage data movement through an internal staging buffer that temporarily holds data during transfer from storage to GPU memory. We configure this staging buffer to be 256 MB. A larger staging buffer allows DirectStorage to batch more requests and maintain a higher queue depth to the NVMe controller, reducing the likelihood of pipeline stalls when the GPU consumes data faster than individual small requests can be submitted. This configuration allows DirectStorage to maintain sustained throughput even when our multiple queues simultaneously request data.

The processing begins with the host system splitting the input file into variable-sized chunks. DirectStorage queues enqueue read requests that transfer data directly into DirectX shared memory buffers, which are simultaneously mapped into CUDA's address space through external memory handles. Once a buffer's fence signals completion, the corresponding CUDA stream launches the parsing kernel. After processing completes, the buffer is recycled for the next chunk if work remains, creating a continuous flow of data through the system.

The CUDA kernel implements a two-level hash table design optimized for the memory hierarchy. Each thread block maintains a private shared memory hash table of $1,024$ entries using FNV-1a hashing[2] for rapid station name lookup. Threads process the input in 128-byte windows to improve memory coalescing. Within each window, threads parse station names and data. Local aggregation into the shared memory hash table reduces contention before a final merge into the global hash table, where atomic operations aggregate the statistics for each unique station.

Once all chunks have been processed and all CUDA streams have synchronized, the global hash table containing the aggregated results resides in GPU memory. We perform a single bulk transfer of this hash table back to CPU memory using `cudaMemcpy`, which represents the only CPU-to-GPU data movement in the entire pipeline. The hash table contains at most $65,536$ entries, and in practice, only a few hundred slots are occupied, given the approximately 400 unique weather stations in the dataset. This means the readback transfer is

negligible compared to the 13 GB input The host then iterates through the hash table, extracts entries with non-zero hash values, and constructs a sorted map of station names to their computed statistics. Finally, the results are formatted according to the challenge specifications and written to the output file using standard file I/O operations. This final write operation occurs entirely on the CPU and is not performance-critical, as it processes only a tiny fraction of the data compared to the input file size.

4 Implementation Details

4.1 Source Code, Executable, and Inputs

All code can be found at https://github.com/devksingh4/jensen-bucks, with instructions to compile and run present in the README file. The compiled executable (with instructions and input files) can be found in the standard "Releases" tab of the GitHub repository.

4.2 Prior Analyses or Transformations

To maintain consistency with the 1BRC specification, we deliberately avoided any preprocessing of input data. The program operates directly on the raw measurement files, precluding optimizations such as dictionary encoding of station names, file compression, or pre-computed metadata. This design choice ensures fair comparison with baseline implementations while introducing the constraint that all parsing and aggregation logic must handle challenge specifications at runtime.

4.3 Major code components

Our implementation consists of three primary components: the DirectStorage I/O pipeline with 16 queues and 32 stream buffers for asynchronous data transfers; two CUDA kernels (`init_global_map_kernel` for hash table initialization and `parse_and_aggregate_kernel` for parsing and aggregation); and a synchronization layer managing D3D12 fences and CUDA events to coordinate the buffer lifecycle. The main processing loop implements a work-stealing pattern where completed buffers are immediately recycled for pending chunks, maintaining sustained pipeline occupancy throughout execution.

4.4 Testing Strategy

To ensure robustness, we adopted an incremental validation strategy that prioritizes correctness across varying input magnitudes. While the 1BRC targets a one-billion-row dataset, we observed that concurrency hazards, such as deadlocks, often manifest in smaller workloads. Therefore, we created a "ground truth" by executing the reference solution against 100 million, 500 million, and 1 billion rows to generate the expected output checksums. We then developed a script to verify our proposed implementation against these baselines. Furthermore, to validate memory safety and thread synchronization, we utilized the NVIDIA Compute Sanitizer[7] to audit every proposed code modification for functional correctness and race conditions using the `racecheck` and `memcheck` tools.

4.5 Hardware Constraints

A significant challenge in optimizing high-I/O programs is distinguishing between software inefficiencies and hardware latency. Our initial testing environment was constrained by a PCIe 3.0 storage drive capped at 2 GB/s read bandwidth, effectively creating a hardware ceiling that masked the benefits of our program optimizations.

Leveraging Capitalism To overcome this challenge, we adopted a two-tiered hardware strategy. First, we migrated to a Samsung 990 Pro SSD, which supports the newer PCIe 4.0 standard[1], to raise the physical baseline of our local tests. Second, to fully emulate the aggregate bandwidth of the official evaluation rig (which features dual NVMe SSDs in a high-throughput configuration[6]), we utilized a RAM disk for testing. This approach removed storage bandwidth as a limiting factor entirely, allowing us to profile the kernel and parsing logic against a virtually unlimited data stream and identify CPU-bound bottlenecks that were previously hidden by slow storage.

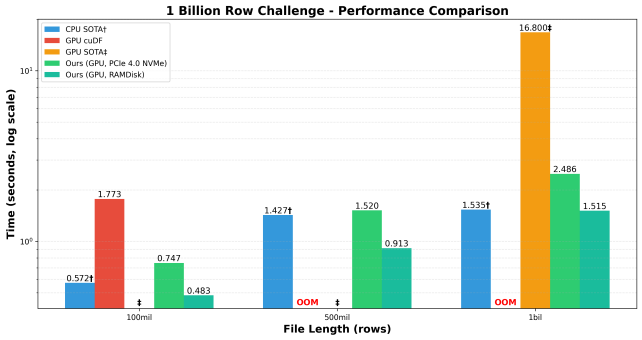
The transition to the new SSD yielded approximately 5 GB/s read throughput, a substantial improvement attributable to PCIe 4.0's doubled bandwidth per lane compared to the previous PCIe 3.0 generation. On the RAM disk configuration, we observed read speeds approaching 8 GB/s. While modern DDR4 and DDR5 memory systems are theoretically capable of sustaining 20-50+ GB/s, our measured ceiling was likely imposed by implementation constraints in the RAM disk software stack or filesystem overhead rather than physical memory bandwidth limitations. Nevertheless, this configuration eliminated storage as a variable entirely, providing a controlled environment to isolate and optimize bottlenecks in our data movement and computation kernels.

4.6 Implementation Status

The current implementation successfully processes the complete 1 billion row dataset with verified correctness. Known limitations include platform hardware/OS specificity and requirements for an NVIDIA GPU with compute capability ≥ 3.5. While we attempted to overcome the Windows-only limitation by using NVIDIA's GPUDirect API on Linux, this API is only supported on datacenter-class GPUs[9]. In contrast, Windows's DirectStorage API works on consumer GPUs with much less stringent requirements[13].

The hash table configuration supports approximately 400, 000 unique stations, far exceeding challenge requirements but representing a theoretical upper bound.

5 Results



† (100mil, 500mil): Locally ran, Ryzen 9 6900HS, (1bil): Official figure reported by the 1BRC repository, 8 cores of AMD EPYC™ 7502P.

‡ (100mil, 500mil): Data unavailable due to incompilable artifact. (1bil): Reported figure in repository.

OOM Out of memory error

File Length (Rows)	100M	500M	1B
CPU SOTA	0.572	1.427	1.535
GPU cuDF	44.1114	OOM	OOM
GPU SOTA	–	–	16.8
Ours (GPU, PCIe4.0 NVMe Drive)	0.747	1.52	2.486
Ours (GPU, RAMDisk)	0.483	0.913	1.515

Table 1. Performance comparison across different file sizes

We choose the evaluate the following five implementations of the 1 Billion Row Challenge.

1. **CPU SOTA**: This is taken from the fastest (CPU) implementation as part of the original 1 Billion Row Challenge, reportedly running on an AMD EPYC™ 7502P, 128GB of DDR4 RAM (speed unknown)[6], and at least 2 NVMe Drives [3].
2. **GPU cuDF**: This implementation is a simple implementation that leverages RAPIDAI's cuDF library in Python. This implementation can be found in this paper's artifact.
3. **GPU SOTA**: This is the reportedly fastest implementation publicly available that utilises the GPU for data processing, reportedly running on a NVIDIA Tesla V100.
4. **Ours (GPU, PCIe 4.0 NVMe)**: This is our implementation executing on a machine with a Ryzen 9 6900HS and a NVIDIA RTX 3070 Ti with a Samsung 990 Pro NVMe PCIe 4.0 SSD.
5. **Ours (GPU, RAMDisk)**: This is our implementation executing on a machine with a Ryzen 5 1600 AF, a NVIDIA RTX 3060 12GB, and 32GB of DDR4-3200 RAM with 16GB allocated towards a RAM disk to store all working files. This is intended to simulate results if we were able to leverage a PCIe 5.0 NVMe SSD.

We evaluate these five implementations on the full 1 billion rows of the 1BRC, a file with the first 500 million rows, and a file with the first 100 million rows.

Notable Observations We were able to beat GPU SOTA, observing a 11.09× speed up when running with a RAM disk, and 6.76× speed up when running with a PCIe 4.0 NVMe SSD. The observed improvement is not only due to algorithmic improvements in the kernel, but also due to our use of the DirectStorage API. The current GPU SOTA employs the classic CUDA `cudaMalloc()` and `cudaMemcpy()` methods to move data to the GPU from the filesystem. The data thus must first be loaded into the CPU's RAM, then to the GPU's VRAM, incurring signficant overhead. Our use of DMA enabled by DirectStorage reduces the transfer overhead, contributing to the observed runtime improvements. This result is also notable as these results were obtained on consumer-grade hardware as opposed to an enterprise-grade V100.

In addition, our implementation reading from a RAM disk beats the runtime of the CPU SOTA on all file sizes. This result is particularly significant given that GPU processing on tasks such as the 1BRC is classically limited by data movement time. Despite this disadvantage, by leveraging an asynchronous batched data transfer approach with DMA to enable and overlapping of computation and memory movement alongside the bypass of CPU RAM. These optimizations substantially mitigate the effects of data movement bottlenecks. Despite these efforts, it is still evident that this problem is still primarily I/O bound. On test runs with a NVIDIA RTX 5090 and the full 1 billion rows, we observe that reported GPU usage reaches no more than 8%.

Our implementation also utilizes VRAM more efficiently. Libraries such as `cuDF` attempt to allocate large contiguous chunks, or in some cases the entire file, into VRAM. When combined with the large size of the 1BRC dataset, this approach renders execution infeasible on conventional hardware. Our streamed implementation only loads a portion of the data into the GPU memory at any given point, resulting in a significantly lower peak memory usage and enabling the program to be run on lower-VRAM machines such as our 3070 Ti.

References

[1] [n. d.]. Samsung 990 PRO PCIe 4.0 SSD. https://semiconductor.samsung.com/consumer-storage/internal-ssd/990-pro

[2] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald E. Eastlake 3rd, and Tony Hansen. 2024. *The FNV Non-Cryptographic Hash Algorithm.* Internet Draft draft-eastlake-fnv-22. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-eastlake-fnv/22/

[3] Hetzner Online GmbH. 2025. Root Server Hardware. https://docs.hetzner.com/robot/dedicated-server/general-information/root-server-hardware#ax-servers. Last changed: 2025-11-14.

[4] Peter Kim. 2024. The One Billion Row Challenge in CUDA: from 17m to 17s. https://tspeterkim.github.io/posts/cuda-1brc.
[5] Lei Mao. 2020. CUDA Stream. https://leimao.github.io/blog/CUDA-Stream/
[6] Gunnar Morling. 2024. The One Billion Row Challenge. https://www.morling.dev/blog/one-billion-row-challenge/
[7] NVIDIA. 2025. compute-sanitizer 13.0.1 documentation. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html
[8] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2025. CUDA, release: 13.0.2. https://developer.nvidia.com/cuda-toolkit
[9] NVIDIA Corporation. 2025. Installation and Troubleshooting Guide — GPUDirect Storage Installation and Troubleshooting. https://docs.nvidia.com/gpudirect-storage/troubleshooting-guide/index.html#cu-file-device-not-supported
[10] RAPIDS Development Team. 2024. cuDF: GPU DataFrame Library. https://github.com/rapidsai/cudf.
[11] RAPIDS Development Team. 2024. libcudf Documentation. https://docs.rapids.ai/api/cudf/stable/libcudf_docs/.
[12] Thomas Wuerthinger, Quan Anh Mai, and Alfonso Peterssen. 2024. CalculateAverage_thomaswue.java. https://github.com/gunnarmorling/1brc/blob/main/src/main/java/dev/morling/onebrc/CalculateAverage_thomaswue.java publisher: Gunnar Morling.
[13] Andrew Yeung. 2020. DirectStorage is coming to PC. https://devblogs.microsoft.com/directx/directstorage-is-coming-to-pc/

A Appendix: An Extended Example

This appendix provides a detailed walkthrough of our code and how data flows from the DirectStorage pipeline through our kernels. Since we divide the input file to be processed into 32MB chunks, we will trace the path of a single chunk in this example for simplicity's sake. This 32MB size is the maximum permitted by DirectStorage.

A.1 Input Data

The 1BRC input data consists of weather station measurements in the following format:

```
San Salvador;18.4
Chihuahua;40.8
Anadyr;-10.8
Warsaw;15.6
Lhasa;2.3
```

Each line consists of a station name followed by a semicolon and a temperature with one decimal place. This means that the input will always be in the format X.Y, XX.Y, -X.Y, or -XX.Y.

A.2 DirectStorage DMA Transfer

A.2.1 Initial Setup. The system initializes 16 DirectStorage queues and 32 total stream buffers during startup. These numbers were chosen to maximize request counts (by increasing queues) while staying within the limited memory of the 3070 Ti Mobile we used for testing (by limiting stream buffers). If we had access to more VRAM, we could increase the number of stream buffers, allowing for more parallel streaming without running out of memory. Additionally, we set the priority of the queue to `DSTORAGE_PRIORITY_REALTIME`, which tells Windows to give our program exclusive and immediate access to the GPU, de-scheduling other Windows processes from the GPU and maximizing the bandwidth and compute provided to our program.

Each of the stream buffers is allocated as a D3D12 shared resource, which makes them accessible to CUDA via external memory importing. This creates a zero-copy path where DirectStorage writes directly to the GPU memory, completely eliminating the CPU from the data transfer path.

A.2.2 Enqueueing Data for Transfer. At the beginning of the program, the system enqueues a single read request for each available stream buffer. These requests are sent round-robin through each of the 16 queues and are serviced as soon as possible. We use fencing to allow for asynchronous tracking of transfer completion for each buffer.

For example, the request for the first chunk would look something like the following:

```
DSTORAGE_REQUEST request = {};
...
request.Source.File.Source = pDsFile.Get();
request.Source.File.Offset = 0;
request.Source.File.Size = 33554432; // 32MB
request.Destination.Buffer.Resource =
    buf->d3d12Buffer.Get();
...
pDsQueues[queueIdx]->EnqueueRequest(&request);
buf->fenceValue = nextFenceValue++;
pDsQueues[queueIdx]->EnqueueSignal(
    buf->d3dFence.Get(), buf->fenceValue);
buf->d3dFence->SetEventOnCompletion(
    buf->fenceValue, buf->fenceEvent);
```

Once these requests are submitted, DirectStorage handles the transfer of the data to the GPU.

A.2.3 DMA Transfer. DirectStorage performs the following operations:

1. **NVMe Command Submission**: Issues optimized NVMe reads directly to the SSD. Since we queue many messages at once and then submit, DirectStorage is able to optimize these reads for maximum bandwidth.
2. **DMA Transfer**: the SSD controller writes data directly to GPU memory using PCIe peer-to-peer
3. **Fence Signaling**: Once the request has been fully serviced and data is available in the GPU, DirectStorage triggers the previously set `EnqueueSignal`, updating the fence value and signaling to the kernel that data is ready to be processed.

A.2.4 Overlap Region Handling. As some chunks may end in the middle of a line, we add an overlap space of 64KB to ensure complete line coverage near chunk boundaries. To prevent duplicate processing of of overlapped regions, threads only read from the start of complete lines, ensuring that each line is processed once across each chunk.

For example, the end of a chunk may look something like

```
...
Dhaka;17.9
Kabul;28.0
Anadyr;3.9
Chia
```

with the next chunk being

```
ng Mai;27.1
Busan;15.2
Tegucigalpa;19.3
Dunedin;15.0
...
```

Without the overlap, this would result in the middle line providing incorrect or corrupt data to our output, violating correctness. With our overlap, we are able to process the full line in the first chunk of the example, preventing this issue.

A.3 CUDA Kernel Processing

Once the data has been written to GPU memory and the fence signals completion of the DMA transfer, the kernel

is launched on the buffer's dedicated CUDA stream. Using streams allows for us to have nonblocking kernel launches and to process multiple buffers at once.

The kernel is launched using a dynamic number of blocks depending on the amount of data in the buffer. This is to allow for better resource utilization for cases where we may not have as much data. For the case of our standard full 32MB chunk, we would launch $33554432/(128 \cdot 256) = 1024$ blocks of 256 threads each.

A.3.1 Thread Work Distribution. Each thread processes a 128-byte window for all the lines of data possible, using a strided access pattern for coalesced memory access and optimized cache usage.

For each line, the thread will do the following:

1. **Hash Station Name**: The thread will extract the name of the station and hash it to get the index in the map where the data should be aggregated
2. **Optimized Temperature Parsing**: Since the format of the temperature data is known, the thread can use a simplified parsing schema and represent the decimal value as an integer, replacing expensive floating point computation with much simpler integer math.
3. **Shared Memory Aggregation**: The thread performs an atomic update with station data into the block's shared memory, which has significantly higher bandwidth than the GPU-wide global memory.

Once the entire block has updated its shared memory, it then consolidates these results into global memory. This allows for fewer global memory accesses in each kernel, reducing the time each kernel spends writing to memory. Since the 1BRC data only has 400 unique stations, this two-level consolidation is key to reducing kernel time.

A.4 Asynchronous Pipelining

While the kernel is processing chunk N, DirectStorage initiates the transfer for chunk N+1, allowing for overlapping data transfer and kernel time. This allows us to mask the most kernel latency, as the data transfer latency is the main bottleneck of the program. The diagram[5] below describes how async processing assists in masking kernel latency.

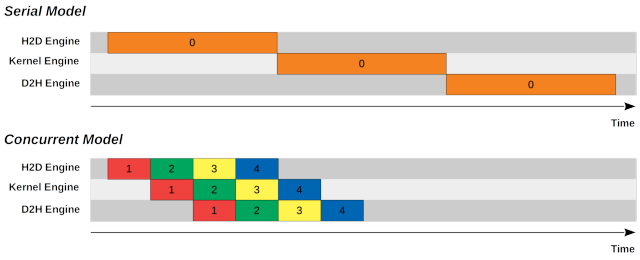


Figure 3. Asynchronous DMA Graph

We see that when compared to serial computation, we're able to hide the kernel latency as we are performing computation during the transfer of the next block.

A.5 Final Aggregation and Output

After all chunks are processed, the global hashmap must be transferred back to CPU memory, after which the CPU reads back results and generates the output file. The output will look something like the following:

```
{Abha=-38.2/18.0/67.8, Abidjan=-22.7/26.0/81.2,...}
```

with each station having min/average/max data. This is the only component done by the CPU (excluding setup), as the rest of the processing of data is performed by the GPU.

A.6 Key Limitations

We enumerate a number of the key limitation of our program below:

1. **Valid Inputs**: As the 1BRC is a challenge that only measures latency, we perform no error handling and expect no malformed data. This is in line with the challenge's parameters, as the input data is consistent and correctly formatted.
2. **Station Name Length**: We hard-code a 100 character limit for station name, so using alternate data with longer names may cause truncation errors.
3. **Windows Only**: The most constraining limitation of our program is that it must be run on very specific hardware systems. DirectStorage is a Windows-only API and is only supported for some Windows machines. The README enumerates the specifications required to run this program. Running the program on an unsupported system may or may not work, and will definitely result in worse performance, as DirectStorage will fall back to traditional data transfer methods.