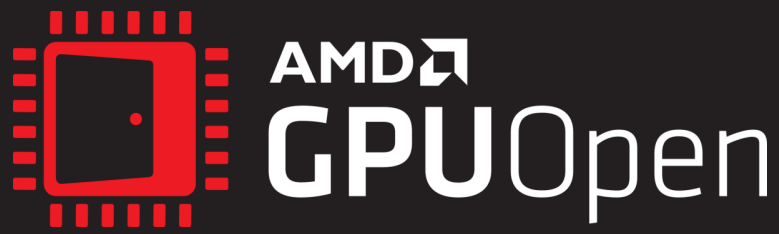
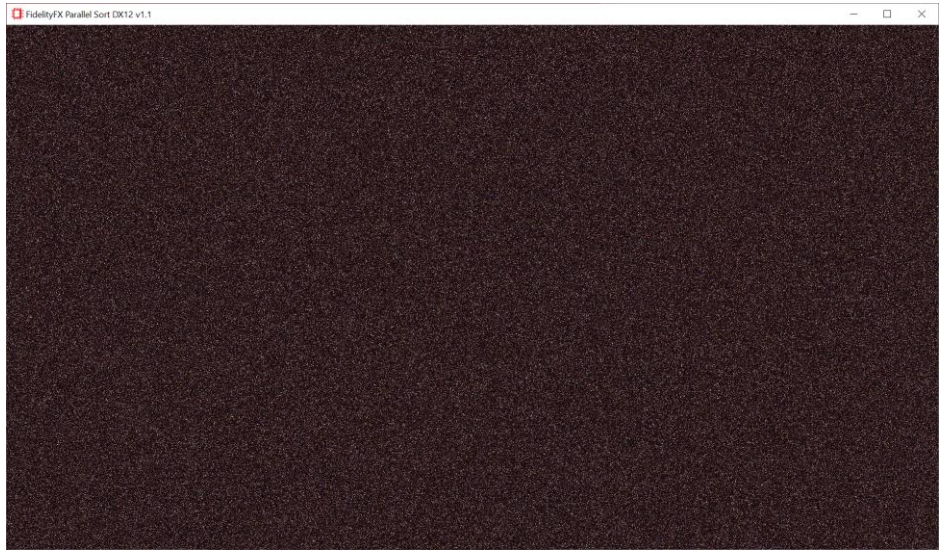


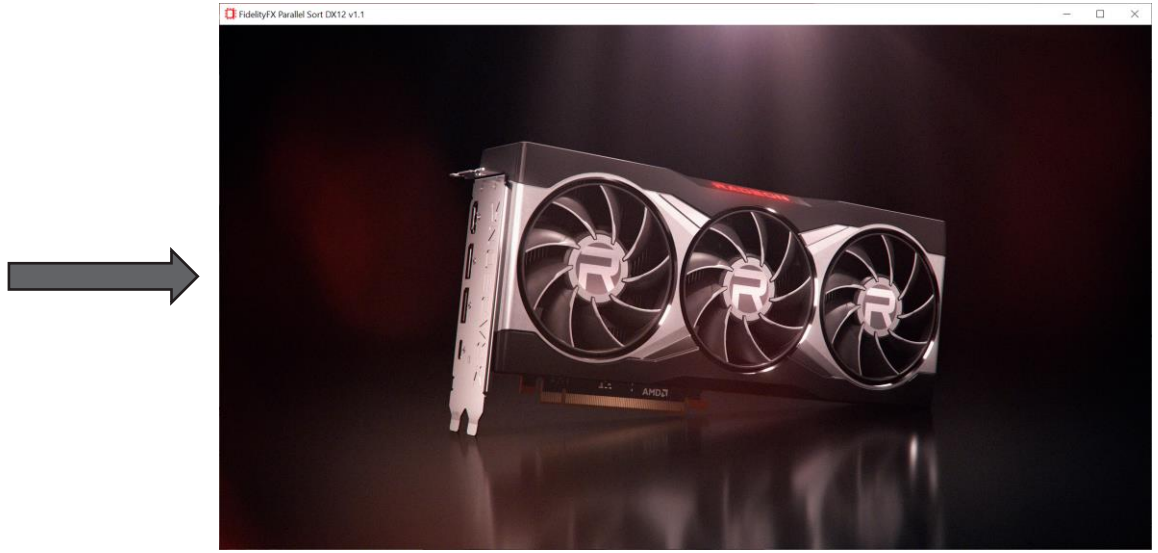
FIDELITYFX PARALLEL SORT

JASON LACROIX, AMD

FIDELITYFX PARALLEL SORT

GPUOpen's FidelityFX Parallel Sort library provides an RDNA-optimized GPU Radix Sort implementation for sorting large data sets quickly



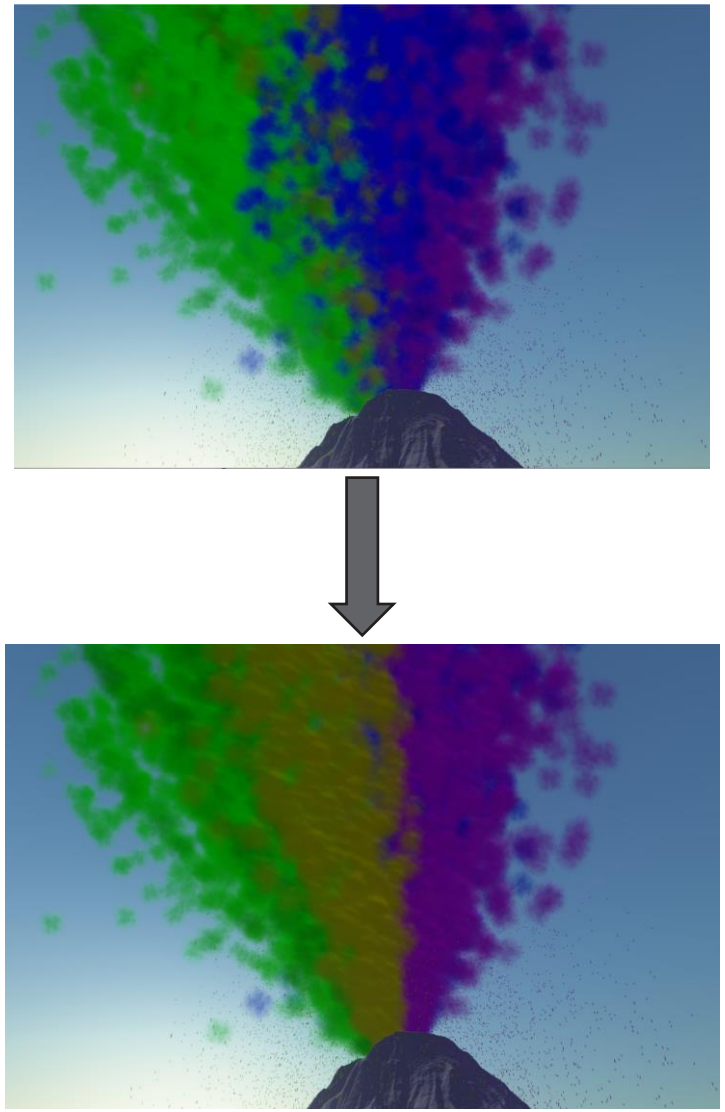


MOTIVATION

In recent years, the number of algorithms that can benefit from a fast-sorting solution have grown. Some of these include:

- GPU-based particle rendering
- Ray sorting for more efficient ray tracing
- Tile sorting-based algorithms (binning, sorted blends, etc.)

Most existing solutions have small upper limits on dataset size



PARALLEL SORT ALGORITHM

ALGORITHM DETAILS

FidelityFX Parallel Sort is based on the Radix sort algorithm

- One of the fastest sorting algorithms, especially for large datasets
- Works with a counting-offset scheme (as opposed to comparisons)
- Data can be sorted in an incremental fashion operating on a different subset in each pass
  - FidelityFX Parallel Sort operates on an incrementing number of 4-bit passes
  - For example, it takes 8 iterations to sort a 32-bit value set

ALGORITHM DETAILS

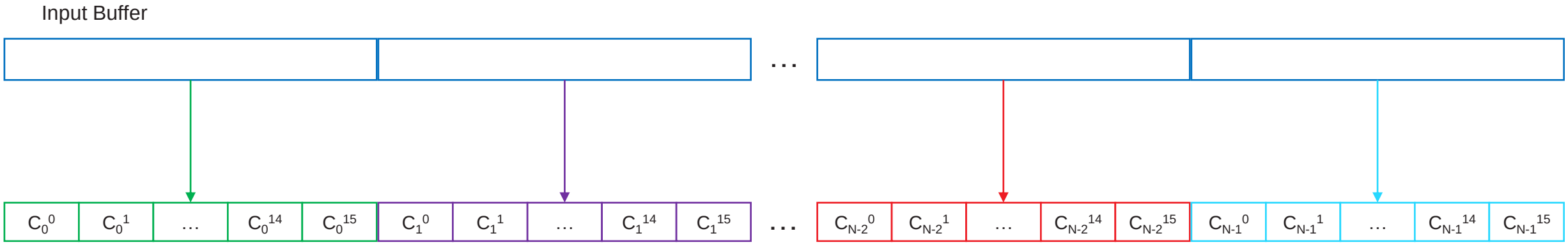
Sorting n bits (example uses 4 bits, as in Parallel Sort)

Start with our input data set distributed equally across all executing threadgroups

ALGORITHM DETAILS

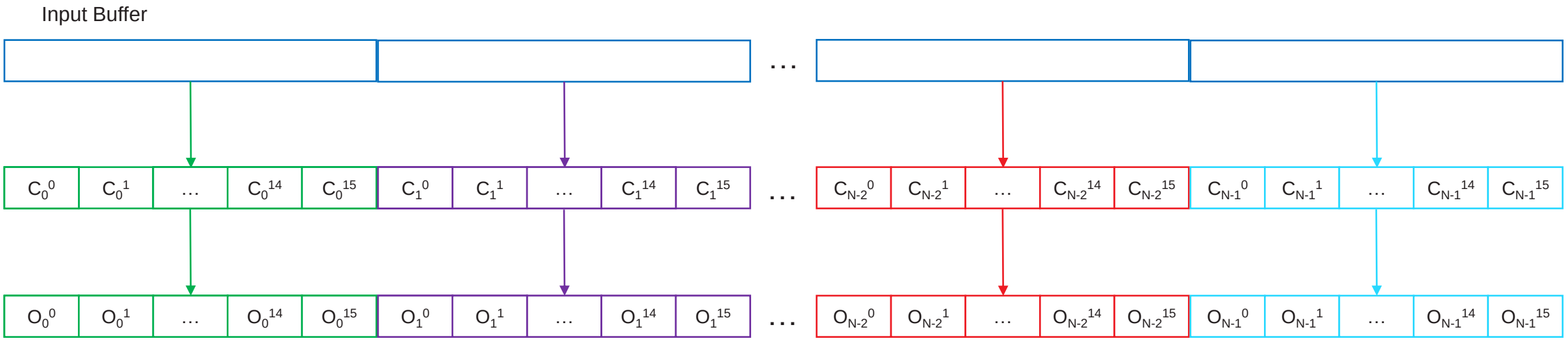
In each threadgroup, count the number of occurrences of each value in its data set

- SourceValue >> (4 bits/iteration * iteration pass) & 0xF



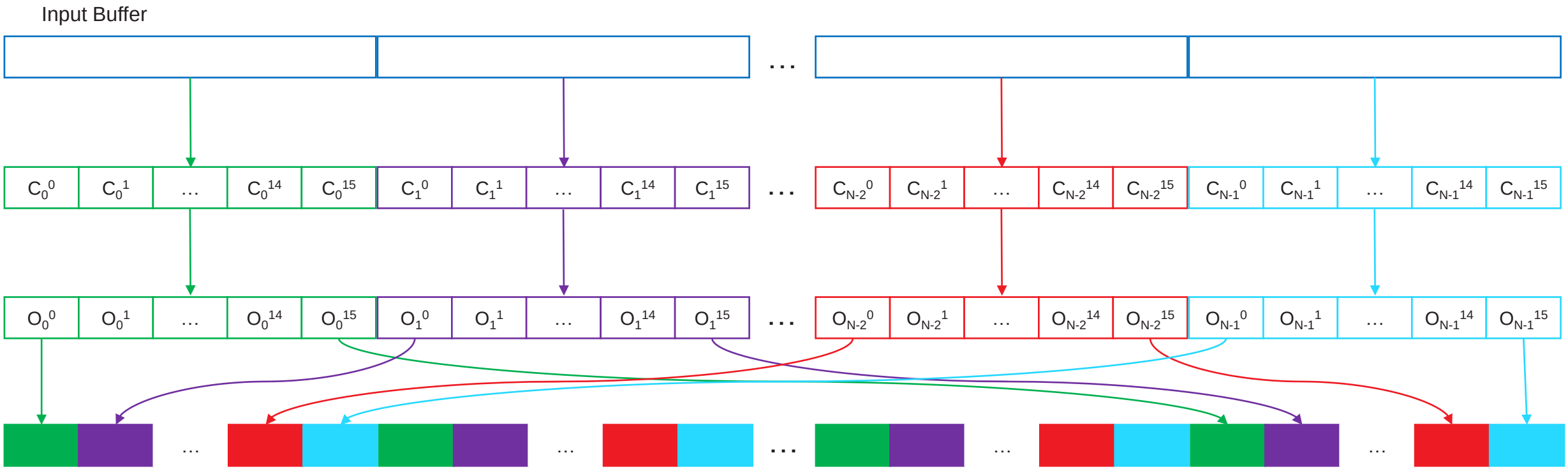
ALGORITHM DETAILS

On each threadgroup, use a prefix scan to generate offsets for each value



ALGORITHM DETAILS

Reorder source values based on calculated offsets



ALGORITHM IMPLEMENTATION

ALGORITHM IMPLEMENTATION

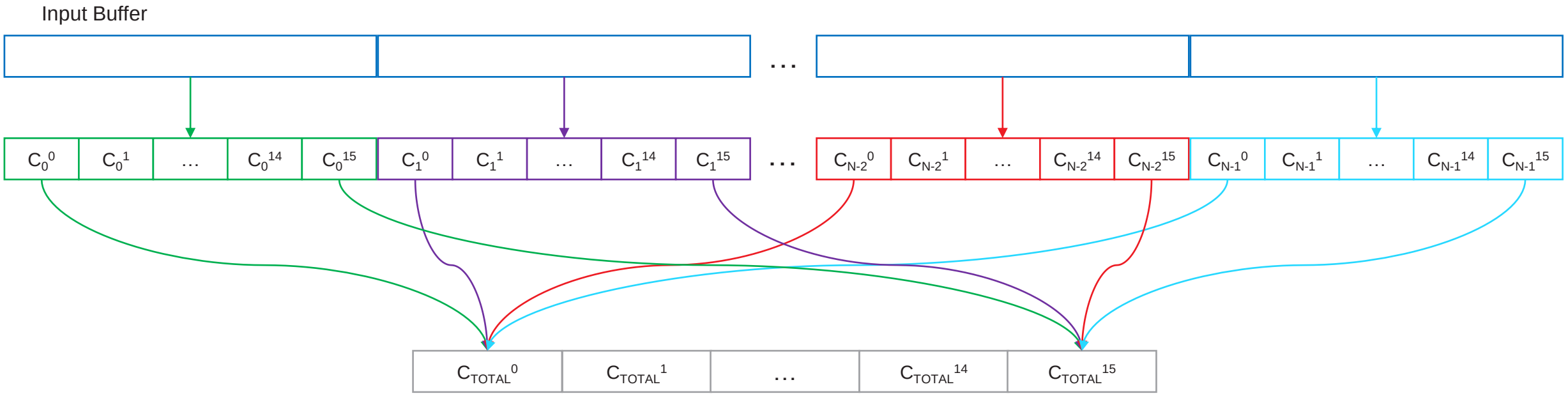
- FidelityFX Parallel Sort operates on blocks of sequential data for optimal reads
  - **Block** size = **4 elements** per thread * **128 threads** per threadgroup
  - Each **threadgroup** will sort 1 or more **blocks**, depending on dataset size
- FidelityFX Parallel Sort goes through a 4-bit sort pass using 4 steps
  - `FFX_ParallelSort_Count_uint`
    - Sample only supports 32-bit uints as of this time
  - `FFX_ParallelSort_ReduceCount`
  - `FFX_ParallelSort_ScanPrefix` (x2)
    - Called once on reduced counts
    - And again on offsets with an additional add with reduced offsets
  - `FFX_ParallelSort_Scatter_uint`
    - Performs value (and payload) re-ordering based on calculated offsets

ALGORITHM IMPLEMENTATION

- `FFX_ParallelSort_Count_uint`
  - Reads in 128 sequential values across all threads simultaneously to load source values
  - Performs `InterlockedAdd` on masked value in LDS to build a histogram of values across the threadgroup
  - Counts across the threadgroup are summed and stored in a SumTable
    - SumTable writes count<X> across all threadgroups sequentially
    - i.e. [$count0_0$, $count0_1$, …, $count0_{NumThreadgroups-1}$, $count1_0$, $count1_1$, …, $count1_{NumThreadgroups-1}$, …]

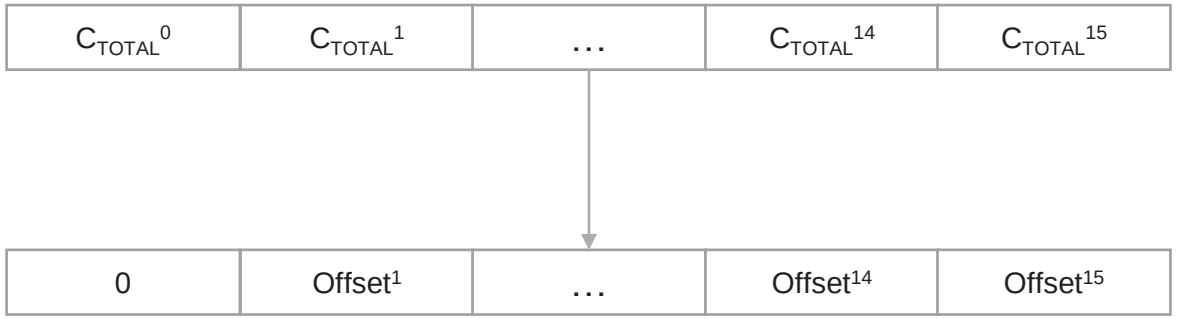
ALGORITHM IMPLEMENTATION

- `FFX_ParallelSort_ReduceCount`
  - Designed to handle large datasets
  - To optimize offset calculations, we reduce the value counts to generate global value counts



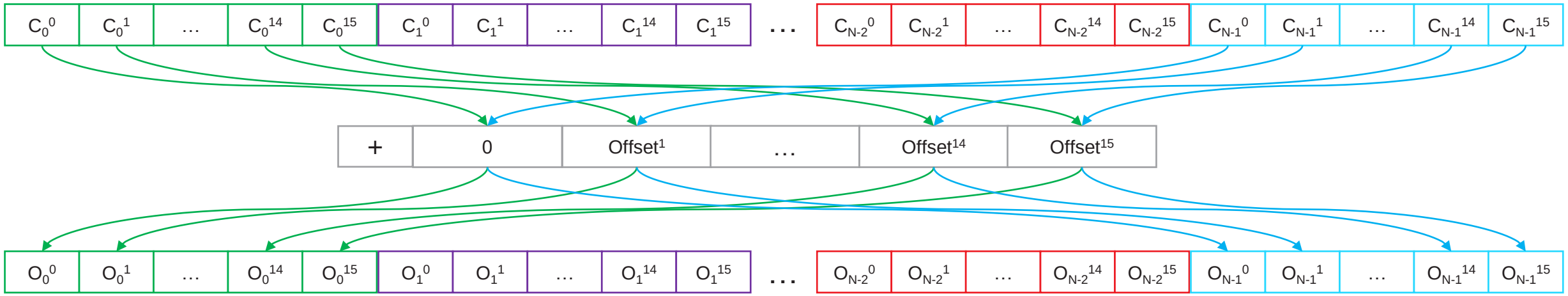
ALGORITHM IMPLEMENTATION

- `FFX_ParallelSort_ScanPrefix`
  - First `ScanPrefix` pass will prefix the count totals into global offsets



ALGORITHM IMPLEMENTATION

- `FFX_ParallelSort_ScanPrefix`
  - Second `ScanPrefix` pass will prefix the threadgroup counts into local offsets
  - Also adds global offsets calculated previously to yield final sorted placement



ALGORITHM IMPLEMENTATION

- `FFX_ParallelSort_Scatter_uint`
  - Re-reads initial count values from the first part of the iteration
  - Performs a local sort of all values in the threadgroup
  - Writes out to the new sorted locations using calculated offsets

INTEGRATION

INTEGRATION - CPU

- Initialization
  - Application must allocate `scratchBuffer` and `reducedScratchBuffer`
  - Use `FFX_ParallelSort_CalculateScratchResourceSize()` to determine the size requirements for the buffers

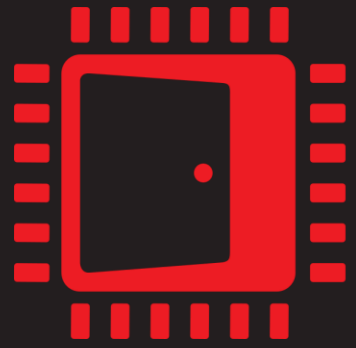
- Other requirements
  - App must provide 2 buffers of adequate size to perform ping-pong sorting of the dataset
    - Doing in-place read/writes is not safe for large jobs and will lead to corruption due to values being moved multiple times (to the wrong location)
  - Constant buffer of type `FFX_ParallelSortCB`

INTEGRATION - CPU

- Run time
  - Populate constant buffer parameters using `FFX_ParallelSort_SetConstantAndDispatchData()`
  - Execute the sort loop 8 times (for 32-bit coverage over 4-bit iterations)
    - See `FFXParallelSort::Sort()` for implementation details

INTEGRATION - GPU

- Create shaders required to call into Parallel Sort shader library
  - `FFX_ParallelSort_Count_uint`
  - `FFX_ParallelSort_ReduceCount`
  - `FFX_ParallelSort_ScanPrefix (x2)`
  - `FFX_ParallelSort_Scatter_uint`
  - Please refer to `ParallelSortCS.hlsl` in the sample



AMD GPUOpen

EPYC

RADEON

RYZEN

AMD

DISCLAIMER & ATTRIBUTION

DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2021 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, RadeonTM and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Vulkan® is a registered trademark of the Khronos Group Inc. DirectX is a registered trademark of Microsoft Corporation. Other names are for informational purposes only and may be trademarks of their respective owners.