# MEMORY-EFFICIENT MULTI-THREADED STREAMING PARTITIONING ALGORITHM

An Undergraduate Research Scholars Thesis

by

ALEXANDER LABBANE

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dmitri Loguinov

May 2022

Major: Computer Science

# RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Alexander Labbane, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

# TABLE OF CONTENTS

# ABSTRACT

Memory-Efficient Multi-Threading Streaming Partitioning Algorithm

Alexander Labbane
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Dmitri Loguinov
Department of Computer Science and Engineering
Texas A&M University

Due to the growth of the modern Internet, data analytics, and cluster computing, massive amounts of data are frequently being generated and need to be processed. In many common data processing applications (e.g., sorting), a set of input keys needs to be partitioned into buckets based on their values. Since key partitioning is an application where data can be processed sequentially (i.e., via streaming), one such programming platform we can use to solve this problem is Vortex. Vortex creates the illusion of an infinite buffer by generating controlled memory access violations that are handled transparently. The buffer can be accessed with a single C/C++ pointer, making Vortex both extremely fast and easy to use.

Efficient parallelization of a key partitioning algorithm is required to take advantage of multi-core processors, which are now found even in low-end consumer hardware. With this in mind, we propose a high-performance, memory-efficient key partitioning algorithm, which makes use of multiple Vortex streams to allow for concurrent partitioning of keys by multiple threads in a single pass over the input data. The resulting algorithm is able to nearly saturate the memory bandwidth of modern Intel Coffee Lake systems and can be applied to develop high-performance, multi-threaded streaming sorts that are capable of utilizing the multiple processing cores available in modern computers.

# ACKNOWLEDGMENTS


## Contributors

I would like to thank my faculty advisor, Dr. Dmitri Loguinov, for his guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my parents for their encouragement, patience, and love.

## Funding Sources

No additional funding was received to support the completion of this work.

# NOMENCLATURE

| | |
|---|---|
| CPU | Central processing unit |
| GB | Gigabyte |
| LSD | Least-significant-digit |
| MB | Megabyte |
| MMU | Memory-management unit |
| MSD | Most-significant-digit |
| RAM | Random access memory |
| SEH | Structured Exception Handling |
| TB | Terabyte |

# 1. INTRODUCTION

Explosive growth of the Internet, cluster computing, and storage technology has led to generation of enormous volumes of information and the need for scalable data computing. One of the central frameworks for handling analysis of such data is MapReduce, which is a programming platform for processing streaming data in external/distributed memory. Despite a significant public effort, open-source implementations of MapReduce (e.g., Hadoop, Spark) are complicated, bulky, and inefficient. Although RAM bandwidth of CPUs is approaching speeds of 100 GB/s, these solutions are only able to achieve data throughput on the order of 100 MB/s. To overcome this problem, we are working on a C/C++ programming abstraction called Vortex that offers a simple interface to the user, zero-copy operation, low RAM consumption, and high data throughput.

An implementation of a fast, parallelized sorting algorithm exists in the form of RADULS2, which is based on an efficient implementation of radix sort [1]. As bucket sizes become small, RADULS2 switches to comparison-based sorting (i.e., insertion sort) and sorting networks [2]. However, RADULS2 is relatively complex and is comprised of more than twenty-thousand lines of source code; in contrast, a single-threaded radix sort implementation using Vortex, named Vortex Sort, is relatively simple using virtual memory primitives in Windows. It has been shown to outperform competing radix sort implementations such as RADULS2 by up to a factor of 3 [3]. However, modern computer processors contain multiple processing cores, even in low-end consumer hardware. This makes efficient parallelization of these algorithms vital. In this research, we focus on developing an efficient multi-threaded key partitioning algorithm that could be used to develop a complete, memory-efficient implementation of a multi-threaded radix sort with Vortex.

## 1.1 Radix Sort

Radix sort is a non-comparison-based sorting algorithm. There are two main flavors of radix sort: least-significant-digit (LSD) and most-significant-digit (MSD).

In a traditional LSD radix sort, a distribution sort is performed based on the LSD of the keys. This involves placing keys in "buckets" based on their LSD [4]. We will call this step partitioning. The buckets are concatenated in order (bucket 0, followed by bucket 1, etc.), and partitioning is repeated on the next least significant digit. This process continues, using the output of the previous iteration as the input to the next iteration, until a final iteration on the MSD places the keys in sorted order.

In an MSD radix sort, the keys are sorted in a similar manner, this time starting with the MSD. Recursively, each bucket is then partitioned using the next most significant digit. This continues until the LSD is processed, at which point the buckets are concatenated together to produce a sorted list of keys.

Regardless of the version of radix sort being used, for in-place implementations, two passes over the data are generally required. The first pass is used to count the size of each bucket, while the second pass distributes the keys into the buckets. Later, we will describe how we can use Vortex streams to eliminate the first pass of data while maintaining an in-place sort (i.e., using only $O(1)$ additional memory).

In general, for short keys (e.g., 32-bit integers), LSD radix sort outperforms MSD due to needing relatively few passes to sort the data, while MSD radix sort may require many recursive function calls. However, for longer keys (e.g., 64-bit integers), MSD radix sort is likely to perform faster because buckets with a small number of elements can be "pruned" and quickly sorted with other approaches, such as sorting networks, which are much faster for small inputs. This optimization cannot be applied to LSD radix sort because buckets are not sorted independent of each other, meaning that the entire input must be processed in each pass.

### 1.2 Virtual Memory

Usually, when we discuss memory in a computer, we are referring to the contents of physical RAM that is installed on the motherboard. However, recall that virtual memory is an abstraction provided by modern operating systems that allows for the separation of logical and physical memory [5]. Regardless of the amount of RAM installed in a system, programmers are given access to an extremely large virtual memory, which we will refer to as the virtual address space. The memory management unit (MMU) dynamically maps virtual memory addresses to physical memory addresses. Note that each process is given its own virtual address space by the operating system (OS), so the memory of one process cannot access or conflict with the memory of another process. This prevents programmers from worrying about the amount of memory available on specific systems or conflicting with the memory of other running processes. Despite the theoretical limit of $2^{64}$ bytes of virtual memory, in modern 64-bit versions of Windows, the virtual address space is a 128 TB range [6].

### 1.3 Paging

Recall that paging is the method used by the OS and MMU to convert virtual memory addresses to physical addresses. A page is a fixed-size, contiguous block of virtual memory that the OS can map to a frame, a fixed-size, contiguous block of physical memory via a page table [5]. In Windows, pages in a process's virtual address space can be in one of three states: free, reserved, or committed [7]. Free pages have not been committed or reserved by the OS and are inaccessible to the process. Pages that have been reserved or committed can be freed with `VirtualFree(MEM_RELEASE)`. Reserved pages are also inaccessible to the process, but virtual addresses in the reserved state cannot be accessed by other memory allocation functions, which is useful to guarantee access to a large contiguous block of virtual addresses. Free pages can be reserved with `VirtualAlloc(MEM_RESERVE)`. Committed pages are pages that have a physical frame associated with them and are accessible by the process. Both free and reserved pages can be committed using `VirtualAlloc(MEM_COMMIT)`, and committed memory can be released while preserving the reservation using `VirtualFree(MEM_DECOMMIT)`. Initially, committed memory is considered committed untouched. Physical memory is not mapped until the program attempts a write to a page that is committed untouched, at which point the memory is committed touched and becomes part of the working set of the process.

When a process attempts access to a page that is not valid in the page table, a page fault is generated by the OS [5]. A page fault can occur when a page that is not resident in page table is accessed or when the process does not have the permissions to access the page. For instance, on first access to a committed page, there is no physical mapping yet, so a page fault is generated. The MMU handles this fault by mapping a new frame and continuing execution of the process at the instruction that generated the fault. By writing a custom exception handler, we can perform custom logic and handle the page faults manually, even if the address of the fault has not been committed yet. Vortex takes advantage of this by generating "controlled access violations, which are intercepted by a custom exception handler that transparently fixes the problem and allows the program to continue" [3].

### 1.4 Vortex Sort

We now summarize the operation of an existing, single-threaded MSD radix sort implementation, which uses a specialized Vortex stream, named Vortex-S [3]. In particular, we will use multiple Vortex-S streams to accomplish the sort.

First, note that each bucket is represented by a single Vortex-S stream. Thus, if we are partitioning by $b$ bits in each step of the sort, $2^b$ streams will be required. In addition, we make the input buffer a Vortex-S stream. Each Vortex-S stream consists of a single virtual buffer that is shared by the data producer and consumer. When the producer attempts to write data to an unmapped section of virtual memory in a stream, a write fault triggers the Vortex-S custom write fault handler, which maps the next block of physical memory to the fault location. When reading from a Vortex-S stream with the consumer, we use guard pages to trigger read faults on block boundaries. The Vortex-S custom read fault handler then unmaps processed blocks of memory and returns them to a shared StreamPool, which is discussed below. Within the buffer of each stream, we map/unmap groups of pages in blocks of size $B$, generally ranging between 1-2 MB to avoid excessive calls to the fault handler, which would harm performance.

**Algorithm 1** Single-Threaded Key Partitioning with Vortex.

```
CACHE_LINE = (1 ≪ CACHE_LINE_BITS);
uint32_t localBuckets[(1 ≪ b) ∗ CACHE_LINE];
short localSize[1 ≪ b];
uint32_t** ptr = buckets;                                  ▷ Vortex stream pointers
function WriteCombine(uint32_t* buf, uint64_t size)
    for (i = 0; i < size; i++) do
        uint32_t buck = (buf[i] ≫ shift) & mask;           ▷ Compute bucket to write key into
        short off = localSize[buck];
        uint32_t* localBuck = localBuckets + (buck ≪ CACHE_LINE_BITS);
        *(localBuck + off) = buf[i];
        localSize[buck] = ++off;
        if off == CACHE_LINE then                          ▷ Temporary bucket is full
            Offload(buck, localBuck);
        end if
    end for
end function
function Offload(uint32_t buck, uint32_t* p)
    __m256i* src = p, *end = src + CACHE_LINE;
    __m256i* dest = ptr[buck];
    while src < end do                ▷ Copy local bucket to Vortex stream using SIMD
        __m256i x = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, x);
        __m256i y = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, y);
    end while
    localSize[buck] = 0;
    ptr[buck] = dest;
end function
```

To maintain an in-place sort, streams need to be able to share and reuse the same physical blocks, leading to the creation of a StreamPool object that is shared by all streams (including the input stream) and maintains a queue of unmapped physical blocks. When the sort is initialized, the StreamPool is allocated enough blocks to store the entire input of size $n$ keys, as well as an additional $2^b$ blocks. The extra blocks ensure there will be enough memory allocated even if the last block in each stream is only partially filled. Thus, the additional memory required by the sort is equal to $B \cdot 2^b$ bytes, which is independent of the input size $n$. Only $O(1)$ additional memory is required, so the sort is in-place. We will adapt this MSD approach to form a multi-threaded partitioning algorithm for an LSD sort that is capable of efficiently partitioning $n$ 32-bit keys by their least significant $b$ bits into $2^b$ buckets (the algorithm can be trivially adapted to partition by the MSD). Assuming all Vortex streams have been created and sufficient memory has been allocated to the StreamPool, Algorithm 1 shows a single-threaded implementation for partitioning using Vortex streams. Keys are first collected into static, local buckets that are small enough to fit in cache before being offloaded to the Vortex streams.

# 2. PARTITIONING

To multi-thread the partitioning algorithm used in Vortex Sort, we must specify an algorithm to distribute work to various threads. In this case, we simply allow working threads to claim 2 MB "jobs" from the Vortex-S input stream, which can be trivially accomplished using interlocked instructions on the buffer pointer. Unfortunately, with multiple threads accessing the input buffer at the same time, the stream can no longer safely unmap blocks and return them to the StreamPool whenever a guard page is triggered. With additional logic, unmapping can be re-enabled on the input stream (making the algorithm in-place), which we will address in future work.

In addition, two potential synchronization issues must be addressed. First, when writing keys into a Vortex stream, there must be some mechanism to ensure multiple threads do not attempt to write into the same location in the buffer. Second, when the fault handler is triggered on a stream, additional logic needs to prevent multiple threads from trying to map physical memory to the same virtual address, which can occur if multiple threads enter the stream fault handler at the same time.

## 2.1 Method 1A

A naïve solution, which we refer to as *Method 1A* (M-1A), involves creating additional Vortex-S streams for each thread to avoid the need for synchronization altogether. In other words, each thread is given its own set of buckets. To write keys into the buckets, threads use Algorithm 1 without modification. If we have $m$ threads, $m \cdot 2^b$ streams are required for the buckets. Since each stream has enough virtual memory reserved to hold the entire input size $n$, this approach requires $n \cdot m \cdot 2^b$ bytes of virtual memory. In addition, if the size of each block is 1 MB, then each stream could waste up to 1 MB of physical memory if the last physical block mapped in the stream is empty. In the case where $n = 64$ GB, $m = 8$, and $b = 8$, this approach will require approximately 130 TB of virtual memory (which exceeds the 128 TB limit set by Windows) and waste up to 2 GB of physical memory. In a data center, a workload such as this is plausible, so in the next section, we explore alternative approaches that prevent the virtual memory usage and physical memory waste from scaling so poorly with the number of threads.

## 2.2 Memory-Efficient Streams

One way to decrease virtual memory usage is to decrease the virtual memory reserved by each Vortex stream. We limit each Vortex stream to a virtual memory reservation size less than or equal to two blocks. This calls for a modified Vortex stream that can operate under this assumption.

We develop a new stream, Vortex-R, that is designed to operate with either one or two blocks of virtual memory reserved. Creating a stream for the case of only one reserved block is quite trivial. When we start writing data into the stream, a page fault triggers the fault handler. The fault handler simply maps a new block. Once the end of this block is reached by the write pointer, we need some mechanism to reset the write pointer to the beginning of the block and tell the stream to unmap the currently mapped block, store a reference to it in a queue local to the stream (so we can remap and read from it later), and map a new empty block. Note that when a block is unmapped, it is not returned to the global queue managed by the StreamPool to prevent other streams from accessing it.

Now, let's consider the case of two contiguous blocks of reserved memory for each stream: block $A$ and block $B$. When the stream is initialized, keys will be written into the buffer at the beginning of block $A$. At some point, the write pointer will move into block $B$, triggering a page fault. In the fault handler, block $B$ will be mapped, and block $A$ will be unmapped, with a reference to the unmapped block stored in a queue for later. Once the write pointer moves beyond block $B$, we require a mechanism to reset it to the beginning of block $A$. Here, the fault handler will proceed as before, this time unmapping block $B$, and mapping block $A$. Thus, by using two blocks per stream, no external communication is required to notify the stream to map a new block. This process continues until all keys are processed and references to all physical blocks have been collected in the stream's queue. To read from the stream, these blocks can simply be remapped later in the same order that they were unmapped.

Regardless of whether we rely on one or two reserved blocks per stream, when the write pointer goes beyond the stream boundary, some mechanism is required to reset it to the beginning of the virtual buffer. Below, we describe several different ways that the write pointer can be reset to the beginning of a stream's buffer, using either one or two blocks per stream.

### *2.2.1 Conditional Statement*

Perhaps the most obvious approach is to reset the write pointer when it is advanced past the beginning of the stream buffer by exactly one block using a conditional statement. Although it would be preferred to handle this within the Vortex stream fault handler somehow, this additional functionality can be implemented with only a small modification to Algorithm 1. While this method is very simple to implement, as shown in Algorithm 2, the branching statement adds non-negligible overhead, resulting in a performance reduction.

**Algorithm 2** Write Pointer Wraparound with Conditional Statement.

```
CACHE_LINE = (1 ≪ CACHE_LINE_BITS);
uint32_t localBuckets[(1 ≪ b) ∗ CACHE_LINE];
short localSize[1 ≪ b];
uint32_t** ptr = buckets;                                ▷ Vortex Stream pointers
function OFFLOAD(uint32_t buck, uint32_t* p)
    __m256i* src = p, *end = src + CACHE_LINE;
    __m256i* dest = ptr[buck];
    while src < end do                 ▷ Copy local bucket to Vortex stream using SIMD
        __m256i x = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, x);
        __m256i y = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, y);
    end while
    if dest - buckets[buck] == blockSize then
        dest = buckets[buck];                            ▷ Reset the write pointer
        streams[buck]->resetBlock();                     ▷ Map a new block
    end if
    localSize[buck] = 0;
    ptr[buck] = dest;
end function
```

### *2.2.2 Structured Exception Handling*

One way to gracefully handle hardware faults in Windows is with Structured Exception Handling, an extension to C++ provided by Microsoft [8]. Using SEH, a custom handler function can be executed whenever an exception occurs. We can use SEH to reset the write pointer whenever it moves past the end of the Vortex virtual buffer and throws an exception, as shown in Algorithm 3. Again, the approach is relatively easy to implement, although SEH incurs some performance penalty as well. We still operate under the assumption of only one reserved block of virtual memory per stream.

**Algorithm 3** Write Pointer Wraparound with SEH.

```
CACHE_LINE = (1 ≪ CACHE_LINE_BITS);
uint32_t localBuckets[(1 ≪ b) ∗ CACHE_LINE];
short localSize[1 ≪ b];
uint32_t** ptr = buckets;                                  ▷ Vortex Stream pointers
function OFFLOAD(uint32_t buck, uint32_t* p)
    __m256i* src = p, *end = src + CACHE_LINE;
    __m256i* dest = ptr[buck];
    while src < end do              ▷ Copy local bucket to Vortex stream using SIMD
        __try                                              ▷ SEH try-except block
            __m256i x = _mm256_loadu_si256(src++);
            _mm256_stream_si256(dest++, x);
            __m256i y = _mm256_loadu_si256(src++);
            _mm256_stream_si256(dest++, y);
        __except(Filter(src, ptr, buck))                  ▷ Call handler on exception
    end while
    localSize[buck] = 0;
    ptr[buck] = dest;
end function

function FILTER(uint32_t src, uint32_t** ptr, uint32_t buck)
    streams[buck]->resetBlock();
    ptr[buck] = buckets[buck];
    src-=1;
end function
```

### *2.2.3 Modular Arithmetic*

Finally, we examine the case of reserving two blocks of virtual memory per stream. Recall that when using two blocks, every time block $A$ is mapped, block $B$ is unmapped and vice-versa. Thus, every time the write pointer moves into a new block, a page fault will be generated that is transparently handled by the Vortex fault handler. To wraparound the write pointer when the end of the virtual buffer is reached, we use the expression

```
ptr = buckStart + (ptr − buckStart) % (2 * blockSize).
```

However, since we can select block size to be a power of two, the expensive modulus operation can be replaced with a bit-wise AND, as shown in Algorithm 4. This remains simple and makes use of only addition and bit-wise operations to reset the write pointer, leading to minimal performance reduction. As such, we proceed using this approach in future sections describing key partitioning with Vortex-R streams.

**Algorithm 4** Write Pointer Wraparound with Modular Arithmetic.

```
CACHE_LINE = (1 ≪ CACHE_LINE_BITS);
uint32_t localBuckets[(1 ≪ b) ∗ CACHE_LINE];
short localSize[1 ≪ b];
uint32_t** ptr = buckets;                                ▷ Vortex Stream pointers
uint32_t buckMod = (-1) ≫ (31 - blockSizePower);
function OFFLOAD(uint32_t buck, uint32_t* p)
    __m256i* src = p, *end = src + CACHE_LINE;
    __m256i* dest = ptr[buck];
    while src < end do            ▷ Copy local bucket to Vortex stream using SIMD
        __m256i x = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, x);
        __m256i y = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, y);
    end while
    dest = buckets[buck] + ((dest - buckets[buck]) & buckMod);
    localSize[buck] = 0;
    ptr[buck] = dest;
end function
```

### 2.3 Method 1B

To create *Method 1B* (M-1B), we modify the multi-threaded key partitioning approach described by M-1A by using Vortex-R streams for the buckets instead of Vortex-S streams; however, each thread still receives its own copy of the buckets to avoid the need for additional synchronization between threads. As such, partitioning remains fast while drastically reducing the virtual memory requirement from $n \cdot m \cdot 2^b$ to $2 \cdot m \cdot 2^b$. Physical memory waste remains the same as M-1A since the same number of streams are still required. In addition, keys are written into buckets by each thread using Algorithm 4, which provides the necessary logic required to wrap the write pointer back to the beginning of the stream.

## 2.4 Method 2A

Next, we will attempt to reduce physical memory usage by allowing multiple threads to share the same buckets in *Method 2A* (M-2A). As such, partitioning will require only $2^b$ buckets, regardless of the number of threads being executed, which reduces physical memory waste to $2^b$ MB if each block is 1 MB in size. We will continue using Vortex-R streams to maintain low virtual memory usage as well. To accomplish this, synchronization is required in two areas: the Vortex-R fault handler and `Offload()`.

### *2.4.1 Multi-Threading Vortex-R Fault Handler*

First, Vortex-R streams need to support multiple threads in the fault handler at the same time. With the current scheme, there are two problems. If multiple threads fault on the same block in a Vortex stream, they will both enter the fault handler and attempt to map a new physical block to the same virtual address. In addition, because Vortex-R streams unmap the previous block every time a fault occurs, it is possible for one thread to unmap a physical block of memory that another thread is still writing keys into.

To combat this, any thread that faults while writing keys into a bucket will pause in the Vortex-R fault handler until *all* threads have faulted in a stream. In practice, this incurs a non-negligible performance penalty since a thread may have to wait a long time for the rest of the threads to fault. To implement this approach, a counter shared by all Vortex streams via the global StreamPool object is used to determine when all threads have reached a fault. Once all threads have faulted, waiting threads are notified to continue by signaling an Event in the StreamPool. In addition, if multiple threads fault on the same stream, a local barrier ensures that only one thread in each stream maps a new block. The resulting synchronization is shown in Algorithm 5.

**Algorithm 5** Multi-Threaded Vortex-R Fault Handler.

```
StreamPool sp;
CRITICAL_SECTION cs[2];                          ▷ Provide mutual exclusion to a block of code
function WRITEFAULT(uint64_t address)
    uint64_t newBlock = 1 - currentlyMapped;
    uint64_t faultCount = InterlockedIncrement(sp.faultedThreads);
    uint32_t index = ((faultCount - 1) / numThreads) & 1;         ▷ Compute which event to use
    if faultCount % numThreads == 0 then
        ResetEvent(sp.events[1 - index]);                      ▷ Make threads wait on next fault
        SetEvent(sp.events[index]);                          ▷ Signal waiting threads to continue
    end if
    EnterCriticalSection(cs[index]);
    if newBlock == 1 - currentlyMapped then▷ Only allow first thread in stream to map block
        LeaveCriticalSection(cs[index]);
        return;
    end if
    WaitForSingleObject(sp.events[index], INFINITE);
    MapBlock(newBlock);
    UnmapBlock(currentlyMapped);
    currentlyMapped = 1 - currentlyMapped;
    LeaveCriticalSection(cs[index]);
end function
```

### *2.4.2 Multi-Threading Write Combine*

Next, `Offload()` must ensure that two threads do not attempt to write to the same location in any of the buckets. This is easily accomplished by using `InterlockedAdd()` to atomically increment the write pointer any time keys are dumped into a Vortex stream, shown in Algorithm 6. Interlocked functions achieve better performance than mutual exclusion objects, so we opt to use them instead whenever possible, especially in functions that are called frequently, such as `Offload()`. In addition, we now perform modulus on `dest` before the copy loop because modulus no longer resets `ptr[buck]` to the beginning of the Vortex stream. This is done to avoid race conditions or the need for additional interlocked functions to reset `ptr[buck]`.

**Algorithm 6** Atomic Increment of Write Pointer with Vortex-R.

```
CACHE_LINE = (1 ≪ CACHE_LINE_BITS);
uint32_t localBuckets[(1 ≪ b) ∗ CACHE_LINE];
short localSize[1 ≪ b];
uint32_t** ptr = buckets;                                   ▷ Vortex Stream pointers
uint32_t buckMod = (-1) ≫ (31 - blockSizePower);
function OFFLOAD(uint32_t buck, uint32_t* p)
    __m256i* src = p, *end = src + CACHE_LINE;
    __m256i* dest = InterlockedAdd(ptr[buck], CACHE_LINE);  ▷ Increment write pointer
    dest = buckets[buck] + ((dest - buckets[buck]) & buckMod);
    while src < end do                ▷ Copy local bucket to Vortex stream using SIMD
        __m256i x = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, x);
        __m256i y = _mm256_loadu_si256(src++);
        _mm256_stream_si256(dest++, y);
    end while
    localSize[buck] = 0;
end function
```

### 2.5 Method 2B

Finally, we introduce *Method 2B* (M-2B) with the goal of eliminating the long wait times experienced by threads in M-2A while maintaining shared buckets between threads. To do so, we revert to using Vortex-S streams to represent buckets. `Offload()` remains almost unchanged from Algorithm 6, although write pointer wraparound is removed. Since previously mapped blocks are not unmapped when writing into Vortex-S streams, threads can immediately map new blocks when they enter the fault handler. This leads to simpler, faster synchronization in the fault handler, shown in Algorithm 7.

**Algorithm 7** Multi-Threaded Vortex-S Fault Handler.

```
StreamPool sp;
CRITICAL_SECTION cs;                              ▷ Provide mutual exclusion to a block of code
function WriteFault(uint64_t address)
    uint64_t index = (address - buf) ≫ sp.blockSizePower;            ▷ Block index to map
    EnterCriticalSection(cs);
    if furthestMappedIndex ≥ index then                 ▷ Support multiple threads in handler
        LeaveCriticalSection(cs);
        return;
    end if
    furthestMappedIndex = index;
    MapBlock(index);
    LeaveCriticalSection(cs);                ▷ Allow all threads to proceed after block is mapped
end function
```

M-2B maintains the same physical memory waste as M-2A (up to $2^b$ MB with 1 MB blocks) and the same virtual memory usage as single-threaded M-1A (up to $n \cdot 2^b$) since virtual memory usage does not increase as the number of threads increases. In addition, the algorithm is faster than M-2A since threads do not have to wait in the fault handler and can immediately map new physical blocks.

# 3. EXPERIMENTS

## 3.1 Setup

Table 3.1: Hardware Configurations

| | $c_1$ | $c_2$ |
|---|---|---|
| CPU | Intel i9-9900k | Intel i7-7820x |
| Platform | Coffee Lake | Skylake-X |
| Cores | 8 | 8 |
| Turbo clock | 5 GHz | 4.5 GHz |
| RAM | 64 GB | 32 GB |
| RAM Type | DDR4-3200 MHz | DDR4-3200 MHz |
| RAM Channels | Dual Channel | Quad Channel |
| OS | Windows Server 2016 | Windows Server 2016 |

Now, we will analyze the performance of the various methods for partitioning keys. Code was compiled using the Microsoft Visual C++ (MSVC) compiler on Windows, although the `WriteCombine()` routine was ported to assembly due to inconsistent performance from the compiled C++ code. Benchmarks were run on two hardware configurations, $c_1$ and $c_2$, which are summarized in Table 3.1. It is important to note that on $c_1$, because the RAM is only dual channel, partitioning becomes bottle-necked by memory bandwidth before all CPU cores are saturated. On $c_2$, however, quad channel memory allows for partitioning speeds to scale until all CPU cores are fully utilized. In all benchmarks, physical memory was mapped in $B = 1$ MB blocks, and $n = 8$ GB of keys were partitioned by their least significant $b = 8$ bits. Results are discussed in the context of these parameters.

In addition, we note that different methods reach peak performance for different values of $CACHE\_LINE$ inside of `WriteCombine()`, which alters the maximum temporary bucket size before keys are offloaded to a Vortex stream. Thus, we first investigate the effect changing the temporary bucket size has on performance as the number of threads is increased.

## 3.2 Temporary Bucket Size

Table 3.2: M-1A Speed Partitioning 8GB on $c_2$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 1018 | 1011 | 943 | 894 |
| 2 | 1981 | 1991 | 1868 | 1608 |
| 3 | 2919 | 2936 | 2764 | 2603 |
| 4 | 3789 | 3827 | 3620 | 3402 |
| 5 | 4577 | 4617 | 4380 | 4175 |
| 6 | 5339 | 5297 | 5117 | 4856 |
| 7 | 6021 | 6097 | 5766 | 5529 |
| 8 | 6549 | 6584 | 6316 | 5950 |

We begin by discussing the effect that the value of $CACHE\_LINE$ has inside of `WriteCombine()`, which controls how many keys accumulate in temporary buckets before they are copied into Vortex streams. Table 3.2 shows how performance scales for M-1A as the value of $CACHE\_LINE$ is adjusted. The benchmark was run on $c_2$ due to its higher memory bandwidth. The performance difference between $2^6$ and $2^7$ keys is negligible, but past $2^7$ keys, performance drops by approximately $5\%$ each time temporary bucket size is doubled. This drop in performance occurs because, for larger temporary bucket sizes, a smaller fraction of the keys are able to fit into the CPU cache. As a result, more cache misses occur when keys are offloaded to Vortex streams. In addition, because no thread synchronization occurs when keys are offloaded in M-1A, there is very little overhead associated with each offload. This makes smaller, more frequent offloads achieve higher performance.

Now, we will examine the effect of temporary bucket size on M-2B, which does require threads to synchronize when keys are offloaded from temporary buckets to Vortex streams. As shown in Table 3.3, in this case, better performance is achieved with temporary bucket sizes of $2^8$ or $2^9$ keys, depending on the number of threads. Because M-2B synchronizes threads when keys are offloaded, the cost of more frequent key offloading is more expensive than M-1A, which outweighs the cache benefits of keeping smaller temporary buckets. For the sake of conciseness, in the following experiments, the highest performing value of $CACHE\_LINE$ will be used to benchmark each method, though data for $CACHE\_LINE$ values ranging from $2^6$ to $2^9$ is available for all other methods on $c_1$ and $c_2$ in Appendix A.

Table 3.3: M-2B Speed Partitioning 8GB on $c_2$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 650 | 832 | 858 | 841 |
| 2 | 1177 | 1548 | 1645 | 1639 |
| 3 | 1656 | 2196 | 2345 | 2379 |
| 4 | 2109 | 2715 | 2995 | 3147 |
| 5 | 2467 | 3230 | 3539 | 3637 |
| 6 | 2813 | 3673 | 4033 | 4177 |
| 7 | 3152 | 4132 | 4551 | 4680 |
| 8 | 3332 | 4534 | 4910 | 5037 |

Finally, on M2-B, recall that the optimal temporary bucket size changes depending on the number of threads that are used. For instance, with a single thread, M-2B perform best with a temporary bucket size of $2^8$ keys. Once the number of threads is increased to $8$, though, the optimal temporary bucket size increases to $2^9$ keys. This happens because, with a small number of threads, there is relatively little competition between threads to write into the shared buckets. As a result, smaller temporary bucket sizes that allow more keys to fit into the CPU cache are able to achieve faster speeds. When more threads are added, however, there is more competition between threads to write into buckets, which increases the cost of synchronization. By increasing the temporary bucket size, this competition is decreased, leading to faster speeds even though a smaller proportion of keys fits into the CPU cache.

### 3.3 Partitioning 32-bit Integers

We now compare the performance of the proposed key partitioning methods on $c_1$ and $c_2$. Results for partitioning 32-bit integer keys on $c_1$ are shown in Table 3.4. First, we note that none

Table 3.4: Speed Partitioning 8 GB on $c_1$ (M keys/s)

| # Threads | M-1A | M-1B | M-2A | M-2B |
|---|---|---|---|---|
| 1 | 925 | 979 | 918 | 891 |
| 2 | 1819 | 1922 | 1753 | 1738 |
| 3 | 2718 | 2837 | 2316 | 2578 |
| 4 | 3540 | 3686 | 2887 | 3341 |
| 5 | 4253 | 4504 | 3438 | 3965 |
| 6 | 4541 | 4607 | 3822 | 4350 |
| 7 | 4507 | 4628 | 4098 | 4339 |
| 8 | 4463 | 4646 | 4130 | 4320 |

Table 3.5: Speed Partitioning 8 GB on $c_2$ (M keys/s)

| # Threads | M-1A | M-1B | M-2A | M-2B |
|---|---|---|---|---|
| 1 | 1011 | 1054 | 912 | 841 |
| 2 | 1991 | 2061 | 1574 | 1639 |
| 3 | 2936 | 3045 | 2381 | 2379 |
| 4 | 3827 | 3940 | 3031 | 3147 |
| 5 | 4617 | 4349 | 3404 | 3637 |
| 6 | 5297 | 5320 | 4093 | 4177 |
| 7 | 6097 | 6166 | 4435 | 4680 |
| 8 | 6584 | 6721 | 4799 | 5037 |

of the methods are able to push much beyond 4.5 billion keys per second in speed, as performance begins to stop scaling past 6 threads. This is indication that the memory bandwidth of $c_1$ is reaching saturation. As shown in Table 3.5, on $c_2$, which has much higher bandwidth quad channel memory, performance continues to increase as more threads are added, with speeds reaching up to 6.7 billion keys per second. This is evidence that, unlike $c_1$, the results on $c_2$ are purely CPU bound and not limited by a lack of memory bandwidth.

On both hardware configurations, M-1A and M-1B achieve noticeably faster results than either M-2A or M-2B. This is because M2-A and M2-B both share Vortex streams between threads, which requires additional synchronization overhead between threads. On the other hand, M-1A and M-1B both create separate Vortex streams for each thread, so no such synchronization is required. Between M-1A and M-1B, we can clearly see that M-1B achieves better performance, and similarly, between M-2A and M-2B, M-2B is faster. Thus, we will perform further comparisons between M1-B and M2-B to describe the benefits and drawbacks of sharing streams between threads.

When utilizing all 8 threads on $c_1$, M-1B is only $\sim 7\%$ faster than M2-B, while the maximum speed delta of $\sim 12\%$ occurs on 5 threads. On $c_2$, the increased memory bandwidth leads to much faster speeds overall; in this case, M1-B outperforms M2-B by a much larger margin. In particular, on 8 threads, M1-B is $\sim 25\%$ faster than M2-B. Because performance is bottle-necked by the CPU rather than memory bandwidth, the extra cost of synchronization between threads becomes much more apparent.

Now, we consider the physical memory usage of the four methods. Recall that each Vortex stream has the potential to waste up to one full physical block of memory in the worst case and will waste one half of a physical block in the average case. Thus, for M-1A and M-1B, since $2^8$ Vortex streams are created per thread, each thread wastes 128 MB of physical memory on average; however, since Vortex streams are shared between threads in M-2A and M-2B, 128 MB of physical memory is wasted total, regardless of the number of threads.

Finally, we discuss virtual memory usage. Since M1-B and M2-A use Vortex-R streams, they reserve only 2 blocks of virtual memory per stream. In M1-B, this leads to $2 \cdot 2^8 = 512$ MB of virtual memory consumption per thread, while M2-A consumes 512 MB of virtual memory total. M1-A and M2-B, on the other hand, reserve the entire input size $n$ for each stream. Thus, when $n = 8$ GB, as in our benchmarks, M1-A consumes $\frac{8 \cdot 2^8 \text{ GB}}{1024} = 2$ TB of virtual memory for each thread, and M2-B consumes 2 TB of virtual memory in total. In our benchmarks, on 8 threads, we calculate M-1A would exceed the 128 TB Windows virtual memory limit partitioning 64 GB of keys, while M2-B surpasses this limit partitioning 512 GB of keys. In contrast, M-2B and M-1A will never exceed the limit since their virtual memory usage does not depend on input size. In practice, virtual memory usage need only be considered when one of the methods requires more virtual memory than is allowed by Windows, as it has negligible impact on resource consumption. For example, only 20 MB of physical memory is needed to reserve 10 TB of virtual addresses [3].

# 4. CONCLUSION

## 4.1 Discussion

Using the Vortex programming model, we have developed several flavors of a multi-threaded key partitioning algorithm, each of which are optimized for either speed or memory consumption. Because the memory management logic is encapsulated in Vortex streams, the resulting multi-threaded partitioning algorithm remains both simple and fast. Futhermore, the algorithms discussed in this paper can be directly applied to develop a multi-threaded radix sort. Like the single-threaded Vortex Sort proposed in [3], a multi-threaded radix sort using the partitioning algorithms discussed here has the potential to outperform competing implementations such as RADULS2 in multi-threaded workloads.

## 4.2 Future Work

Of the various methods discussed in this paper, M1-B provides the fastest speeds, while M2-A and M2-B offer the highest memory efficiency. Further development of these algorithms may lead to discovery of a new method, M3, which achieves partitioning speeds comparable to M1-B while retaining the memory efficiency found in M2-A and M2-B. Such an algorithm would be an optimal choice in all use cases, regardless of the available hardware resources.

In addition, to achieve in-place key partitioning, the Vortex-S read fault handler should be altered to support the use of multiple threads reading from the stream at the same time. This would allow the memory used by the input buffer to be unmapped as keys are processed, making the partitioning in-place with no additional modification to the partitioning algorithm.

Finally, the algorithms described in this paper can be further developed into a multi-threaded version of Vortex Sort, and performance can be compared to the fastest in-place sorts currently available, such as RADULS2 and the single-threaded Vortex Sort.

# REFERENCES


[1] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, "Sorting data on ultra-large scale with raduls," pp. 235–245, 04 2017.

[2] M. Kokot, S. Deorowicz, and M. Dlugosz, "Even faster sorting of (not only) integers," in *ICMMI*, 2017.

[3] C. Hanel, A. Arman, D. Xiao, J. Keech, and D. Loguinov, *Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications*, p. 623–638. New York, NY, USA: Association for Computing Machinery, 2020.

[4] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd ed.* Addison-Wesley, 1998.

[5] A. Silberschatz, P. B. Galvin, and G. Greg, *Operating System Concepts, 9 ed.* Wiley, 2013.

[6] A. Viviano, "Virtual address spaces." [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/memory/page-state.

[7] "Page state." [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces.

[8] "Structured exception handling (c/c++)." [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-170.

# APPENDIX A: PARTITIONING SPEED

Table A.1: M-1A Speed Partitioning 8GB on $c_1$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 1089 | 1076 | 977 | 925 |
| 2 | 2150 | 2123 | 1927 | 1819 |
| 3 | 3172 | 3112 | 2872 | 2718 |
| 4 | 4086 | 4032 | 3733 | 3540 |
| 5 | 4442 | 4525 | 4421 | 4253 |
| 6 | 4405 | 4474 | 4529 | 4541 |
| 7 | 4362 | 4427 | 4508 | 4507 |
| 8 | 4241 | 4365 | 4398 | 4463 |

Table A.2: M-1B Speed Partitioning 8GB on $c_1$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 1060 | 1073 | 979 | 914 |
| 2 | 2078 | 2071 | 1922 | 1824 |
| 3 | 2975 | 3071 | 2837 | 2667 |
| 4 | 3925 | 3903 | 3686 | 3547 |
| 5 | 4493 | 4389 | 4504 | 4073 |
| 6 | 4485 | 4624 | 4607 | 4534 |
| 7 | 4491 | 4612 | 4628 | 4579 |
| 8 | 4459 | 4569 | 4646 | 4469 |

Table A.3: M-1B Speed Partitioning 8GB on $c_2$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 1043 | 1054 | 983 | 936 |
| 2 | 1992 | 2061 | 1927 | 1831 |
| 3 | 2939 | 3045 | 2832 | 2672 |
| 4 | 3860 | 3940 | 3712 | 3481 |
| 5 | 4720 | 4349 | 4295 | 4284 |
| 6 | 5459 | 5320 | 5276 | 4742 |
| 7 | 6027 | 6166 | 5933 | 5351 |
| 8 | 6532 | 6721 | 6632 | 6238 |

Table A.4: M-2A Speed Partitioning 8GB on $c_1$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 874 | 984 | 945 | 918 |
| 2 | 1447 | 1748 | 1752 | 1753 |
| 3 | 1950 | 2411 | 2249 | 2316 |
| 4 | 2107 | 2890 | 3032 | 2887 |
| 5 | 2366 | 3385 | 3421 | 3438 |
| 6 | 2296 | 3577 | 3707 | 3822 |
| 7 | 2502 | 3464 | 3878 | 4098 |
| 8 | 2313 | 3386 | 3894 | 4130 |

Table A.5: M-2A Speed Partitioning 8GB on $c_2$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 671 | 875 | 918 | 912 |
| 2 | 1196 | 1598 | 1707 | 1574 |
| 3 | 1657 | 2152 | 2452 | 2381 |
| 4 | 1755 | 2714 | 3066 | 3031 |
| 5 | 1956 | 2970 | 3610 | 3404 |
| 6 | 2443 | 3389 | 4007 | 4093 |
| 7 | 2340 | 3576 | 4018 | 4435 |
| 8 | 2485 | 3801 | 4303 | 4799 |

Table A.6: M-2B Speed Partitioning 8GB on $c_1$ (M keys/s)

| # Threads | Temporary Bucket Size | | | |
|---|---|---|---|---|
| | $2^6$ keys | $2^7$ keys | $2^8$ keys | $2^9$ keys |
| 1 | 859 | 955 | 919 | 891 |
| 2 | 1481 | 1708 | 1715 | 1738 |
| 3 | 2083 | 2441 | 2510 | 2578 |
| 4 | 2597 | 3086 | 3217 | 3341 |
| 5 | 3028 | 3603 | 3845 | 3965 |
| 6 | 3485 | 3978 | 4302 | 4350 |
| 7 | 3780 | 4233 | 4247 | 4339 |
| 8 | 4002 | 4176 | 4211 | 4320 |