Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Applications

Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov

Internet Research Lab (IRL)
Department of Computer Science and Engineering
Texas A&M University, College Station, TX, USA 77843

March 19, 2020
Agenda

• Introduction
• Motivation
• Producer Consumer
• Partitioning and Sorting
• Experiments
Introduction

• Streaming is a commonly employed paradigm for data-intensive computing
  — Often, traditional streaming applications and software packages are unsuited for extreme performance, or rates close to the speed of hardware
  — Moreover, data streaming continues to offer the same block-based communication model of the 1950s
  — Because of this, programmers must choose between “fast, but complex” (e.g., hand-tuned assembly), and “simple, but slow” (e.g., Apache Hadoop) solutions for large problems
Introduction

- Many applications in data analytics, information retrieval, and cluster computing process massive amounts of information.

- In this paper we introduce the Vortex programming model, which has the following goals:
  - Offer a simple abstraction for larger-than-RAM inputs.
  - Squeeze maximum performance out of hardware.

- Usually, these are conflicting goals, but we show that this does not have to be the case.
  - Vortex leverages access violations to create the illusion of an infinite buffer in user space.
  - It is by far the simplest to use and fastest platform for various streaming workloads.
Agenda

• Introduction

• **Motivation**

• Producer Consumer

• Partitioning and Sorting

• Experiments
Motivation: Coding Simplicity

• Consider the task of finding a user-defined string in a long (e.g., 32 TB) stream of data using `strstr()`
  — The stream could be originating from a disk, another thread, or arriving in real time from the network

• Traditional block-based solution:

```c
Search (char* str, uint64 size, uint64 blockSize)
    strlen = strlen(str); buf = new char[blockSize];
    pos = 0; bufStart = 0;
    while (not end of data) do
        size = blockSize - 1 - pos;
        bytes = GetNextBlock(buf + pos, size);
        buf [pos + bytes] = NULL;
        if ((ptr = strstr(buf, str)) != NULL) then
            return ptr - buf + bufStart;
            bufStart += bytes + pos - (strlen – 1);
            pos = strlen – 1;
        memcpy(buf, buf + blockSize – 1 – pos, pos);
```

— Error-prone pointer calculations
— `Memcpy()` for data crossing block boundaries
— Tedious coding practice, slow development
Motivation: Coding Simplicity

• Instead, we would like a much simpler memory abstraction that allows treating streams as infinite:

```c
Thread Producer (char* buf, uint64 len)
    memset(buf, ‘a’, len);
    buf [len] = NULL;

Thread Consumer (char* buf)
    return strstr(buf, “Hello World!”);
```

• Ideally, this abstraction would provide:
  — Coding simplicity
  — No Memcpy() or boundaries
  — No error-prone pointer management
  — Complete transparency, including synchronization
  — Ability to make large (e.g., 32 TB) memset()/strstr() calls
Motivation: Faster Iterator Abstractions

- Consider implementing a producer-consumer pipeline between threads

```
Thread Producer (Iterator* it, uint64 len)
    for (i = 0; i < len; i++)
        do
            it->Write(i);

Thread Consumer (Iterator* it, uint64 len)
    for (sum = 0, i = 0; i < len; i++)
        do
            x = it->Read(); sum += x;
```

- Commonly, this is done with an iterator abstraction
- Iterators greatly reduce programming effort
- Error-prone block management is abstracted away

Let’s look further into iterators!
Motivation: Faster Iterator Abstractions

- **Iterators** exhibit non-trivial overhead
  - **Writer**: 3 loads and 3 stores
  - **Reader**: 4 loads and 2 stores
- **An optimal solution** requires 1 load and 1 store
  - Iterators thus unnecessarily stress the L1 cache, which can become a huge bottleneck in certain applications

**Iterator Internals**

```cpp
Iterator::Read()
    x = bufR [posR];
    if posR == blockSize - 1 then
        empty.push (bufR);
        bufR = full.pop ();
        posR = 0;
    else
        posR++;
    return x;
```

```cpp
Iterator::Write(int x)
    bufW [posW] = x;
    if posW == blockSize - 1 then
        full.push (bufW);
        bufW = empty.pop ();
        posW = 0;
    else
        posW++;
```
Motivation: Faster Iterator Abstractions

- The desired abstraction would allow memory to be processed uninterrupted (i.e., without boundaries or explicit synchronization)

  - Benefits of this approach:
    - Requires 1 load and 1 store per item
    - Depending on CPU, may be 2-4x faster than an iterator
    - Regular pointers are abstracted as being “infinite” (i.e., not constrained by physical RAM)
    - Could help to maximize application throughput

```
Thread Producer (int* buf, uint64 len)
  for (i = 0; i < len; i++) do
    buf [i] = i;

Thread Consumer (int* buf, uint64 len)
  for (sum = 0, i = 0; i < len; i++) do
    sum += buf [i];
```

May be larger than RAM (e.g., 32 TB)
Motivation: Non-Counting In-Place Partitioning

- Consider partitioning \( n \) keys across \( k \) arrays (e.g., during radix sort)
  - The size of each output buffer is unknown a-priori, which generally requires a counting pass to pre-allocate buffers
  - Key movement either requires \( 2n + O(1) \) memory or needs slow iterator abstractions
- It is desirable to eliminate these constraints and
  - Distribute the keys without the histogram pass
  - Operate in-place (i.e., using \( n + O(1) \) total memory)
  - Achieve close to optimal speed
Agenda

• Introduction
• Motivation
• **Producer Consumer**
• Partitioning and Sorting
• Experiments
Virtual Memory in User Space

- General idea behind Vortex
  - Access to reserved, uncommitted virtual memory generates a page fault
  - These faults result in exceptions that can be caught by a user-space handler
  - We can thus cause controlled, sequential-access violations in virtual memory
  - To fix the violation, we map physical pages to the location of the fault in the stream
  - Once the memory is available, we transparently restart the read/write instruction that caused the fault
Vortex-A

- To avoid faulting per 4-KB page, operations proceed in units (blocks) of size B (e.g., 1-2 MB)
  - To allow out-of-order reads, let $M$ be the consumer comeback, i.e., the number of blocks by which it can return to reprocess the data
- Threads are synchronous - the producer is invoked per-block within the fault handler

And so on

Producer is invoked in the handler

$M=1$, return block to OS
Vortex-A

- Drawbacks of this model:
  - The abstraction is non-transparent to the producer thread, and thus incoming data must still be produced in block-sized increments rather than continuously.
  - Threads are necessarily synchronous, as the producer is only invoked once the consumer encounters a fault.
  - Consumer comeback is handled by $M$, but producer comeback has no such accommodation.
Vortex-B

• In this model, the producer is not aware of the existence of an underlying stream
  ─ Instead, the producer writes into an infinite buffer
  ─ Adds producer write-ahead N and comeback L control

• Threads are asynchronous
  ─ Achieved by tracking and limiting the consumer via guard pages, which cause access violations

• Employs the classical bounded producer-consumer solution to track empty and full blocks

Let’s see it in action!
Vortex-B

- Operation with $M = 0$, $L = 0$, $N = 2$

Consumer faults on a guard page

Producer fault!

The producer installs a guard page and must wait for the consumer $N=2$.

And so on $M=0$, return LRU block to OS.
**Vortex-B**

- **Drawbacks of this model:**
  - Blocks cannot be safely consumed until they are protected by guard pages, and thus the minimum distance between threads is the full size of a block.
  - Producer and consumer threads share a virtual buffer, making it more difficult to isolate them (e.g., forward consumer jumps are not supported).
  - Instead of maintaining a pre-allocated stack of blocks, memory is obtained from and released to the OS, which incurs a severe performance penalty.
Vortex-C

- Further improves upon Vortex-B
  - Pre-allocates physical memory at the start of the program instead of during runtime
  - Unlike the previous models, gains speed by retaining blocks for remapping rather than freeing to the OS
- Instead of using guard pages to track the consumer, this method uses dual-buffers
  - Threads get separate virtual-memory buffers for runtime address space isolation
  - Blocks are quickly remapped between streams

Let’s see it in action!
**Vortex-C**

- Operation with $M = 1, L = 1, N = 1$

Producing

Producer fault!

```
| Block 0 | Block 1 | Block 2 | Block 3 | Block 0 | Block 1 | Block 2 | Block 3 | Block 0 |
```

Consumer fault!

Consuming

A full block is mapped to the consumer buffer

```
| Block 0 | Block 1 | Block 2 | Block 3 | Block 0 | Block 1 | Block 2 | Block 3 | Block 0 |
```

And so on with the Vortex

The consumed block is then remapped to the producer
Agenda

• Introduction
• Motivation
• Producer Consumer
• **Partitioning and Sorting**
• Experiments
Partitioning and Sorting

• We adapt Vortex to create a novel variant of **bucket sort** which utilizes:
  ─ Non-counting data partitioning to avoid a histogram pass
  ─ In-place data shuffling to stream sort with \( n + O(1) \) RAM

• To achieve non-counting data partitioning
  ─ Each sort bucket is reserved to the full size of input

• The result is the first in-place streaming radix sort
  ─ Posts a **2-4x performance improvement** over prior work
  ─ Provides **out-of-place speeds** with in-place operation

• Finally, this abstraction **does not require** specialized code or memory management to achieve in-place sorting, being instead **totally transparent**
Agenda

- Introduction
- Motivation
- Producer Consumer
- Partitioning and Sorting
- Experiments
## Available Test Configurations

<table>
<thead>
<tr>
<th>Hardware</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel 3930K</td>
<td>Intel 4930K</td>
<td>7820X</td>
</tr>
<tr>
<td>Platform</td>
<td>Sandy Bridge</td>
<td>Ivy Bridge</td>
<td>Skylake-X</td>
</tr>
<tr>
<td>Test drive</td>
<td>24-disk RAID</td>
<td>24-disk RAID</td>
<td>M.2 SSD</td>
</tr>
</tbody>
</table>
Experiments

### File I/O Speed (MB/s)

<table>
<thead>
<tr>
<th>Framework</th>
<th>c₁ Read</th>
<th>c₁ Write</th>
<th>c₃ Read</th>
<th>c₃ Write</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::fstream</td>
<td>43</td>
<td>88</td>
<td>51</td>
<td>140</td>
<td>8%</td>
<td>2 MB</td>
</tr>
<tr>
<td>Win. MapViewOfFile</td>
<td>69</td>
<td>147</td>
<td>1,161</td>
<td>*</td>
<td>8%</td>
<td>32 GB</td>
</tr>
<tr>
<td>Linux mmap</td>
<td>1,892</td>
<td>1,170</td>
<td>1,917</td>
<td>641</td>
<td>3%</td>
<td>30 GB</td>
</tr>
<tr>
<td>Vortex-A</td>
<td>2,235</td>
<td>1,547</td>
<td>1,272</td>
<td>651</td>
<td>8%</td>
<td>5 MB</td>
</tr>
<tr>
<td>Vortex-B</td>
<td>2,231</td>
<td>2,394</td>
<td>3,211</td>
<td>650</td>
<td>8%</td>
<td>5 MB</td>
</tr>
<tr>
<td><strong>Vortex-C</strong></td>
<td><strong>2,238</strong></td>
<td><strong>2,399</strong></td>
<td><strong>3,266</strong></td>
<td><strong>674</strong></td>
<td><strong>1%</strong></td>
<td><strong>5 MB</strong></td>
</tr>
</tbody>
</table>

Vortex-C is 1.7x faster than mmap

### Batched Producer-Consumer Rate (GB/s)

<table>
<thead>
<tr>
<th>Framework</th>
<th>Two core</th>
<th>All cores</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c₁</td>
<td>c₂</td>
<td>c₃</td>
</tr>
<tr>
<td>Apache Storm</td>
<td>1.7</td>
<td>1.4</td>
<td>2.4</td>
</tr>
<tr>
<td>Naiad</td>
<td>2.7</td>
<td>3.1</td>
<td>4.4</td>
</tr>
<tr>
<td>Queue of Blocks</td>
<td>6.4</td>
<td>7.3</td>
<td>11.4</td>
</tr>
<tr>
<td>Vortex-B</td>
<td>4.3</td>
<td>4.4</td>
<td>4.6</td>
</tr>
<tr>
<td><strong>Vortex-C</strong></td>
<td><strong>13.5</strong></td>
<td><strong>16.4</strong></td>
<td><strong>23.3</strong></td>
</tr>
</tbody>
</table>

Vortex-C is 5-10x faster than Storm
# Experiments

## Populating an 8 GB Vector on c₃ (GB/s)

<table>
<thead>
<tr>
<th>Framework</th>
<th>Memory</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Untouched</td>
<td>Pre-Faulted</td>
<td></td>
</tr>
<tr>
<td>std::vector</td>
<td>0.7</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>RUMA rewired vector, 4 KB pages</td>
<td>-</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>RUMA rewired vector, 2 MB pages</td>
<td>-</td>
<td>14.3</td>
<td></td>
</tr>
<tr>
<td>Chained Blocks</td>
<td>6.8</td>
<td>18.8</td>
<td></td>
</tr>
<tr>
<td><strong>Vanishing Array (Vortex-S)</strong></td>
<td><strong>25.1</strong></td>
<td><strong>25.1</strong></td>
<td></td>
</tr>
<tr>
<td>Static Buffer</td>
<td>8.0</td>
<td>28.5</td>
<td></td>
</tr>
</tbody>
</table>

Vortex-S is 3.1x faster than an untouched static buffer.

Vortex-S reaches 88% of static buffer speed.

## Partitioning Speed of 8 GB on c₃ (M keys/s)

<table>
<thead>
<tr>
<th>Framework</th>
<th>Write</th>
<th>Combine</th>
<th>k=256</th>
<th>k=512</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-pass</td>
<td>N</td>
<td></td>
<td>339</td>
<td>322</td>
</tr>
<tr>
<td>Chained blocks</td>
<td>N</td>
<td></td>
<td>450</td>
<td>413</td>
</tr>
<tr>
<td>Vortex-S</td>
<td>N</td>
<td></td>
<td>492</td>
<td>445</td>
</tr>
<tr>
<td>Pre-allocated buckets</td>
<td>N</td>
<td></td>
<td>509</td>
<td>464</td>
</tr>
<tr>
<td>2-pass</td>
<td>Y</td>
<td></td>
<td>364</td>
<td>344</td>
</tr>
<tr>
<td>chained blocks</td>
<td>Y</td>
<td></td>
<td>461</td>
<td>449</td>
</tr>
<tr>
<td>Vortex-S</td>
<td>Y</td>
<td></td>
<td>607</td>
<td>523</td>
</tr>
<tr>
<td>Pre-allocated buckets</td>
<td>Y</td>
<td></td>
<td>637</td>
<td>567</td>
</tr>
</tbody>
</table>

Applied to partitioning, Vortex-S achieves 92-96% static buffer speed.
## Experiments

### Fastest In-Place Radix Sorts (M keys/sec)

<table>
<thead>
<tr>
<th>Sort Type</th>
<th>Year</th>
<th>8 GB of keys</th>
<th>24 GB of keys</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB Radix</td>
<td>2014</td>
<td>19</td>
<td>23</td>
<td>26</td>
<td>18</td>
<td>21</td>
<td>26</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>MSB Radix</td>
<td>2017</td>
<td>24</td>
<td>25</td>
<td>32</td>
<td>24</td>
<td>26</td>
<td>32</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>MSB Radix</td>
<td>2019</td>
<td>17</td>
<td>19</td>
<td>26</td>
<td>25</td>
<td>30</td>
<td>39</td>
<td>26</td>
<td>32</td>
</tr>
<tr>
<td>Vortex Sort</td>
<td>2020</td>
<td>71</td>
<td>84</td>
<td>127</td>
<td>68</td>
<td>80</td>
<td>121</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fastest Out-Of-Place Radix Sorts (M keys/sec)

<table>
<thead>
<tr>
<th>Sort Type</th>
<th>Year</th>
<th>8 GB of keys</th>
<th>24 GB of keys</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB Radix</td>
<td>2011</td>
<td>25</td>
<td>25</td>
<td>39</td>
<td>25</td>
<td>26</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB Radix</td>
<td>2014</td>
<td>24</td>
<td>26</td>
<td>42</td>
<td>24</td>
<td>26</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB Radix</td>
<td>2016</td>
<td>19</td>
<td>23</td>
<td>34</td>
<td>19</td>
<td>23</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB Radix</td>
<td>2017</td>
<td>25</td>
<td>29</td>
<td>41</td>
<td>25</td>
<td>29</td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB Radix</td>
<td>2017</td>
<td>44</td>
<td>58</td>
<td>67</td>
<td>44</td>
<td>58</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vortex Sort</td>
<td>2020</td>
<td>71</td>
<td>84</td>
<td>127</td>
<td>68</td>
<td>80</td>
<td>121</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Speedup Factor of Vortex-S

<table>
<thead>
<tr>
<th>Compared to</th>
<th>8 GB</th>
<th>24 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c₁</td>
<td>c₂</td>
</tr>
<tr>
<td>Best in-place</td>
<td>2.9</td>
<td>3.3</td>
</tr>
<tr>
<td>Best out-of-place</td>
<td>1.6</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Vortex is **2.9-4.0x faster** at sorting 8 GB than the nearest in-place radix sort competitors. Vortex is **2.7-3.1x faster** at sorting 24 GB than the nearest in-place radix sort competitors. Even considering out-of-place sorts, Vortex is still **1.6-1.9x faster** at sorting 8 GB, and can run sort sizes **twice as large**.
Thank you!
Any questions?

Contact: Carson@cse.tamu.edu