Supporting input dependent access pattern algorithms on GPUs using GPUfs

Sagi Shahar
Technion - Israel Institute of Technology
sagi@tx.technion.ac.il

Mark Silberstein
Technion - Israel Institute of Technology
mark@ee.technion.ac.il

ABSTRACT

Accelerating processing of very large datasets on GPUs is challenging, in particular when algorithms exhibit unpredictable data access patterns. In this paper we investigate the utility of GPUfs, a library that provides direct access to files from GPU programs, to implement such algorithms. We analyze the system's bottlenecks, and suggest several modification to the GPUfs design, including new concurrent hash table for the buffer cache and a highly parallel memory allocator. We evaluate our changes by implementing a real image processing application which creates collages from a dataset of 2 Million images. The enhanced GPUfs design improves the application performance by 2× over the original GPUfs and outperforms both 12-core parallel CPU and standard CUDA-based GPU implementations, while significantly simplifying GPU application design.

1. INTRODUCTION

Discrete GPUs are commonly used for speeding up a variety of data processing applications. However, accelerating computations on large data sets which exceed GPU physical memory is a challenge. In fact, many algorithms that are known to be highly efficient on GPUs for small data sets, such as kNN search [7], do not scale to real-world data.

Computations on large data sets necessarily involve file accesses, but current GPUs cannot access a host file system directly because they lack file system access support. Therefore, an application developer needs to coordinate GPU accesses to secondary storage via explicit application-level management code running on a CPU, which performs file accesses on GPU's behalf and manages low level data transfers to/from GPU memory. Furthermore, all the data that a GPU may need must be resident in the GPU memory prior to computations, and it is the responsibility of a GPU developer to ensure that this is the case. As a result, all the potential GPU accesses to data must be known *before* the GPU execution starts. This requirement impedes the use of GPUs to run data processing algorithms with irregular data access pattern on large datasets.

Recently, GPUfs [10] introduced file system (FS) support for GPUs. By providing a standard FS API to GPU code, GPUfs enables processing of large datasets by reading files from a running GPU kernel on demand, thereby eliminating the need to know all future GPU data accesses in advance and obviating CPU-side data management code.

These features make GPUfs ideal for implementing applications with input-dependent data accesses, like image collage. The application takes a single image as an input, breaks it into blocks, and then replaces each block with a visually similar image from a large dataset. To quickly find the matching image from a multi-million image dataset, the algorithm employs a Locality Sensitive Hashing (LSH) [3] heuristic, which enables to narrow down the search to only a few images. In particular, the algorithm computes the LSH of each block in the original image which in turn identifies a subset of images in the large dataset that are then exhaustively searched through. Therefore, the accesses to the dataset are input-dependent and cannot be predicted without computing the LSH first. The collage application is representative of a large family of LSH-based algorithms used in many areas dealing with large datasets, such as image similarity search, image classification, or related tweets search.

In this work we examine the applicability of GPUfs for implementing large-scale applications with unpredictable data access pattern. We show that the original GPUfs design had a number of important limitations which lead to significant degradation in the application performance. We identify the main bottlenecks and suggest a few design modifications. In particular, we dramatically improve the parallel performance of the GPUfs buffer cache by replacing the traditional radix-tree implementation with a concurrent hash map. We also optimize the GPUfs dynamic memory allocator, and introduce batching of memory transfers from the host to enable 4K pages in the GPUfs buffer cache.

We evaluate the performance of the enhanced GPUfs layer by implementing the image collage application. We show that the new version achieves an average of 2× speedup over the original GPUfs. Moreover it is up to 30% faster than a 12-core run of a multithreaded CPU implementation using Intel's Thread Building Blocks, and 10% faster than a native GPU implementation without GPUfs.

In summary, this paper makes the following contributions.

1. We provide an in-depth analysis of the GPUfs design limitations in applications with a complex data access pattern.

2. We introduce a new concurrent GPU hash map, a concurrent memory allocator and a host-device batching mechanism, and integrate them into the GPUfs design

3. We show significant performance improvements on realistic workloads over CPU and native GPU implementations, as well as over the original GPUfs version.

The rest of the paper is structured as follows. Section 2 describes the related work on processing large datasets on

GPUs. Section 3 provides an overview of the LSH algorithm, and the way it is used in the collage application. Section 4 analyzes the impact of the LSH input-dependent memory access pattern on the system performance, highlighting the main GPUfs bottlenecks. We then describe our modifications to the original GPUfs design in Section 5 and present our initial results in Section 6.

2. RELATED WORK

The problem of automatic and efficient GPU data management has been addressed in several recent works, however handling input-dependent data accesses from GPU to files is the novel contribution of this paper.

Several approaches have been proposed to handle datasets larger than the GPU physical memory. Some of them are application-specific, like Shredder [1] while others, such as PTask [9] require using a special programming model. Both of these approaches use data chunking and multiple kernel invocations to achieve their goal. However, static data chunking requires advanced knowledge of the data that will be accessed at each stage of the algorithm. Therefore, for many structured applications such as dense matrix operations or streaming applications, data chunking works well. Yet, this approach is difficult to apply for applications that exhibit input-dependent data accesses, like the one used in this paper.

Lee at al. [6] propose to predict an application data access pattern by combining static analysis of GPU kernel code, and execution of an inspection kernel. This inspection kernel is a modified version of the original GPU kernel leaving only the code responsible for computing data access locations. This mechanism, however, cannot handle complex data-driven accesses.

NVIDIA recently introduced a Unified Virtual Memory mechanism that allows both CPU and GPU to share data using a unified address space. The memory is allocated on the device and is moved transparently between the GPU and the CPU on demand. Even though this mechanism may be used to store our dataset and let the GPU driver handle the necessary data transfers, it has two major limitation in the context of our application that accesses files. (1) Data access from a CPU to the buffer located in Unified Memory can not be performed while a kernel is running. In particular, in order to access data from a file, a GPU kernel still needs to be broken into several separate kernels, where the file accesses are performed by a CPU between kernel invocations. (2) The maximum size of the Unified Memory buffer is limited because this buffer is allocated on the GPU. Therefore its size cannot exceed the physical memory size of the GPU. This limitation only allows us to work on datasets that can reside entirely inside the GPU memory and is inapplicable to processing large datasets, which is the main focus of this work.

Similarly to the Unified memory concept by NVIDIA, Region-based Virtual Memory for GPUs was proposed by Ji at al. [5]. Here, the buffers are allocated on the host, therefore the allocation size is no longer limited to the size of GPU physical memory. Further, data can be accessed from the host while the GPU kernel is running. This approach, however, still requires the entire dataset to be copied into a dedicated memory location on the host prior to GPU execution and does not support direct file system access from GPU.

3. BACKGROUND

We provide a brief description of the LSH algorithm, its use in the collage application, and current GPUfs design.

3.1 The Locality Sensitive Hash algorithm

Locality Sensitive Hash-based algorithms is a family of algorithms used mainly for retrieval of data with high dimensionality from very large datasets. These algorithms use a locality-preserving property of certain hash functions in order to map *neighboring* objects in the high dimensional space into the same *bucket* in a hash table. Given a query, an algorithm computes its LSH hash, and retrieves all the data points in the corresponding bucket. Since LSH preserves a certain locality metric, the retrieved data points correspond to approximate nearest neighbors of the query. This property makes LSH particular useful for a variety of applications that require sub-linear data retrieval from large datasets, like image similarity search [11], related tweets search [8] and image classification [7].

Algorithm 1 LSH based Approximate Nearest Neighbor

```
for each query do                          ▷ Can be done in parallel
    Extract features
    for i ← 1, numKeys do
        k ← calculate LSH key
        for each item in bucket[k] do
            Insert to shortList            ▷ Remove duplicates
        end for
    end for
    for each item in shortList do
        Read from file
        Calculate distance
        Update minimum
    end for
end for
```

The use of LSH in these applications follows a common structure depicted in Algorithm 1.

1. Perform feature extraction for a given query and compute its LSH hash keys. Multiple keys to multiple hash tables are performed to improve precision. This stage can be done independently for each query, requires no access to the actual dataset, and its compute intensity can vary depending on the feature extraction method.

2. Fetch candidate data points from the dataset based on their LSH keys computed in (1). Each LSH key corresponds to a small subset of the dataset, but the subsets corresponding to different keys are not disjoint. Therefore, after the candidates are fetched, they are merged into a single shortlist while removing duplicate data points.

3. Compare query and the data points from the shortlist. This is often the most compute intensive part of the algorithm since the list may contain several dozens of data points.

From the data access perspective, the second stage of the algorithm performs accesses into the dataset based on the keys computed in the first stage. Therefore, the actual data accesses are input-dependent and cannot be predicted in advance. Moreover, these algorithms are applied to very large

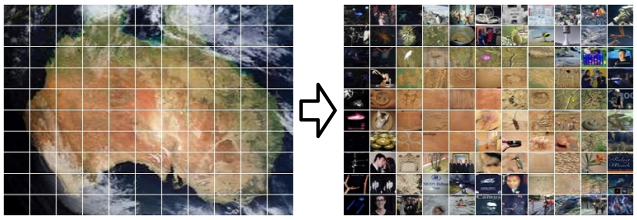

Figure 1: Collage example

datasets that do not fit even into CPU memory, and certainly exceed the even more limited GPU memory, making prefetching of the complete data set into CPU or GPU memory impossible.

3.2 Image collage application

In this paper we use an LSH based, approximate nearest neighbour search as a building block for an image collage application.

The application accepts as input an image from the user. For each 32×32 pixels block in the image, we search for a *similar* image inside a 2 Million images dataset. The dataset itself is stored in a file as a continuous array of images, while each image location is 4KB aligned.

We create 32 hash tables (using different LSH functions), where each hash table bucket contains the file offsets of images corresponding to that bucket. The application is implemented as follows. In the first stage we compute the LSH keys of each of the 32x32 pixels blocks in the input image, based on the block color histogram. Then for each one of those keys we gather the locations of candidate images in the dataset. In the last stage we read the candidates from disk and perform the short list comparison to find the best match. An example image created by the collage application is presented in Figure 1.

3.3 GPUfs

In this work we implement all the stages of the LSH algorithm on GPU, and use GPUfs [10] file system layer in order to perform data accesses in the second stage of the algorithm. GPUfs exposes a POSIX-like file system API to GPU applications. The high level design is presented in Figure 2. GPUfs manages its own buffer cache inside GPU memory in order to minimize redundant accesses to the host file system if the data has been already accessed before. The buffer cache is managed in pages of the same size, configured at system initialization. If the requested page is not found in GPU memory, GPUfs accesses the host file system by issuing a Remote Procedure Call that is handled by the CPU.

4. PERFORMANCE ANALYSIS

We evaluate three versions of the LSH collage algorithm.

1. Multithreaded CPU version implemented using Intel's Thread Building Blocks (TBB)
2. Native GPU version which comprises two GPU kernels for Step 1 and Step 3 of the algorithm, and uses CPU to create the shortlist and read data from files.
3. GPUfs version which is a implemented as a single kernel for all the stages of the algorithm and uses GPUfs for file system accesses.

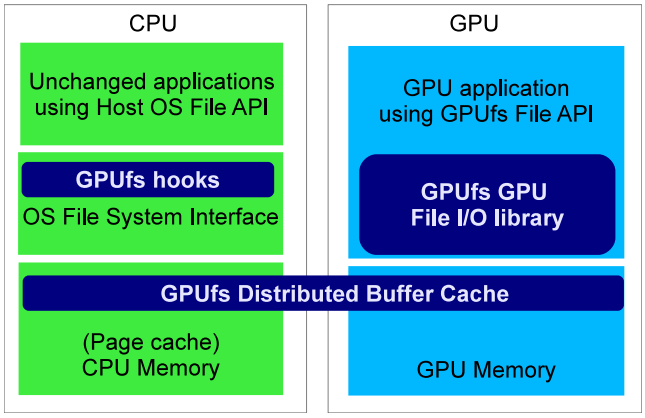

Figure 2: GPUfs Architecture (reprinted from [10])

Both the CPU and GPUfs versions follow a straight-forward approach to access data from files. Specifically, after finding the candidate images in Stage 2, they make these images available to Stage 3 by issuing *mmap* (*gmmap*) from the dataset. If different LSH keys correspond to the same file, the respective call will not access the file system again, instead returning the page that has been already accessed before. Thus, these implementations take advantage of the file system layer to eliminate redundant accesses to the disk.

In contrast, the native GPU version cannot access the files from the GPU and therefore requires pre-processing in order to explicitly eliminate duplicate images in the shortlist. Specifically, the CPU first gathers the indices of all the candidates, eliminates duplicates by using a hash table, reads all the images into a staging area and transfer the data to the GPU. Finally, CPU code builds the list of offsets to each image and to which shortlist each image belongs. Importantly, this implementation will only work so long as all the images in the shortlist fit into the GPU physical memory. Larger inputs which will use longer shortlists will require a significantly more complex implementation.

We evaluate all the three implementations by generating collages using the Tiny Images dataset. We use a subset of the dataset containing 2 million images each 32×32 pixels (7.6GB in total). As an input we use several images of increasing sizes ranging from 512×384 (192 blocks) to 4096×3072 (12288 blocks). We run on a SuperMicro server featuring $2 \times$ 6-core Intel i7-4960X CPUs at 3.6GHz with 15MB L3 cache per CPU, and an NVIDIA GeForce GTX TITAN GPU with 6 GB of GDDR5 memory. We run Ubuntu Linux kernel 3.13.0-32, with CUDA SDK 7.0 and NVIDIA GPU driver 346.29. In this experiment GPUfs is configured to use 4K pages in its buffer cache, in order to match the feature sizes of images in the file system. To reason about the performance of the software implementation we eliminate the disk access latency and store the data in CPU RAM using RAMfs. Optimizing the disk access performance from GPU is left for future work. We run each test ten times, using the first 3 times as a warm up. We report an average of the last 7 iterations.

As can be seen in Figure 3, while the CPU implementation outperforms the GPU native implementation for smaller input images, the performance gap decreases for larger images. For the largest image with 12288 blocks, the GPU

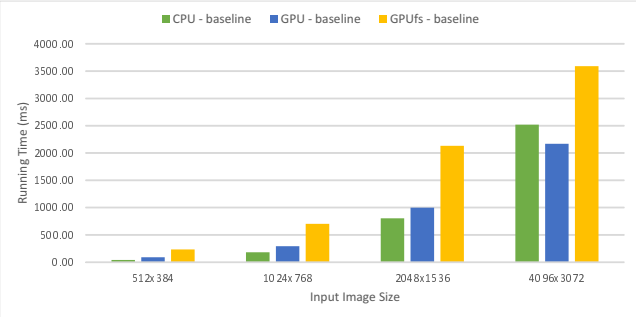

Figure 3: Performance analysis of the collage application on images of different sizes.

implementation is faster by 16% compared to the CPU. The GPUfs implementation, however, is significantly slower than both CPU and GPU implementations across all inputs, with an average slowdown of 2.2× versus the native GPU implementation, and 3.4× slowdown versus the CPU version.

These results warrant in-depth analysis of the GPUfs performance which we describe in the next subsection.

5. GPUFS ANALYSIS & OPTIMIZATIONS

In this section we evaluate the performance of the GPUfs system, focusing on its three major components.

The first component is the GPUfs buffer cache management mechanism. Due to a highly concurrent nature of GPU applications, this mechanism is required to support high levels of concurrent accesses. As was shown by Clements at al. in their work on RadixVM[2], the traditional lock based radix tree implementation used in the buffer cache can suffer from high contention and limit the overall performance significantly. We show that GPUfs suffers from similar problem and show a more efficient design.

The second component is the internal GPUfs memory allocator. In our applications, multiple threads will read new data into the file system and therefore require allocations of new pages in the cache, forcing the allocator to handle many concurrent requests. This component turns out to be a major bottleneck.

The third component that we evaluate is the CPU/GPU memory transfer mechanism. Since our application reads large amounts of data from the disk and performs a relatively small amount of compute per read data, the memory bandwidth of CPU-GPU transfers becomes a problem.

As we show in this section, each of these bottlenecks can potentially hide the others. For example, the contention on Radix Tree locks hide the contention on the memory allocator. Therefore, evaluating the performance of each part requires isolating different components which is tricky in a complex real system.

5.1 Improving Radix Tree concurrency

In order to evaluate the overhead of the Radix Tree we measure the amount of time the implementation spends on radix tree searches in the buffer cache. As can be seen in Table 1, the radix tree data structure implementation suffer from heavy contention. The implementation spends more than 95% of the radix tree search time on a global spinlock, so it runs almost entirely in a serial fashion and eventually

Image size	Total search time	Spinlock time
512x384	191.87ms (81.90%)	187.36ms (79.97%)
1024x768	622.45ms (88.6%)	607.59ms (86.46%)
2048x1536	1919.01ms (89.97%)	1855.82ms (87.01%)
4096x3072	3087.75ms (85.99%)	2914.18ms (81.16%)

Table 1: Radix tree search time. The number in parentheses is the percentage of the total running time

Figure 4: Theoretical buffer cache miss rate for images of different sizes.

takes up to 90% of the overall running time.

The original implementation was designed for workloads with a high ratio of radix tree reads versus inserts. Therefore, the radix tree was originally implemented with lock-free reads but with coarse grain locking to protect against concurrent tree modifications. However in the collage workload, the insertion rate is much higher than the one the system was design for.

To estimate the insertion rate we measure the buffer cache miss rate. In particular, we estimate the miss rate by calculating the amount of data reuse for the algorithm. The data reuse is defined as the ratio between the number of unique images read and the total number of images read. Assuming an infinite cache size and a page size of 4K, this number will yield the buffer cache miss rate.

As Figure 4 shows, the miss rate decreases significantly as we increase the size of the input image. This happens because all the queries are derived from the same image. Given the images smoothness we can see that as we use images with higher resolution, neighbouring blocks in the image will map to similar areas with similar histograms and therefore access similar images in the dataset. Even for smaller images where we only access a small portion of the dataset, we can see a miss rate of less than 50%.

However, comparing these miss rates to the ones exhibited by the workload with which GPUfs has been originally evaluated, like matrix-vector product and approximate matching, we see that the LSH algorithm exercises accesses with much lower data reuse, hence much higher insertion rate. Even for large images with the miss rate as low as 13.28%, about every 8^{th} access to the buffer cache will result in an insertion and will lock the entire buffer cache radix tree.

Solution: concurrent hash table

We replace the radix tree based implementation with a concurrent hash table. The table resolves file offsets to pages holding file data in GPU memory. The total size of the hash

Image size	Total search time	Memory allocator time
512x384	43.27ms (79.39%)	37.34ms (68.51%)
1024x768	167.82ms (85.01%)	147.43ms (74.68%)
2048x1536	585.13ms (83.09%)	469.54ms (66.67%)
4096x3072	845.18ms (62.25%)	495.30ms (36.48%)

Table 2: Hash table search time using old memory allocator. The number in parentheses is the percentage of the total running time (Not including IPC transfers)

table is set to be 16 times of the total number of available pages that fit in the GPU memory allocated to the buffer cache. Using this configuration, the memory overhead of the hash table is less than 5% and uses and additional 100MB in order to manage 2GB of memory. The theoretical collision rate in this case is 3% even when the buffer cache is almost full.

We implement a concurrent hash table using fine-grain locking per bucket for insertion and lock-free for reads. We also considered a lock-free lazy list implementation [4], but we decided against it. In the lazy list each insertion or removal requires two locks, which improves concurrency for hash tables with high collision rate. In our case, however, these locks will most likely lock the entire bucket anyway since the average size of the bucket is less than 2. In our implementation we require only a single lock, and provide a similar level of concurrency.

In order to evaluate the hash table performance under high contention from multiple simultaneous requests, we replace the CPU-GPU data transfers with empty stubs. We generate the maximum achievable number of simultaneous requests from the system for the LSH application which stand at 200K inserts per seconds.

As can be seen in Table 2, the new hash table implementation is much faster, and the overall access time to the buffer cache is reduced by 4× on average compared to the original numbers in Table 1.

5.2 Improving memory allocator concurrency

Looking at the leftmost column in Table 2, we can see that while the contention on the buffer cache itself is much lower, this optimization exposes a new bottleneck which was previously hidden. As long as the system suffered from high contention when modifying the radix tree, the memory allocator was not a bottleneck, but it becomes one with the faster concurrent hash table design. The original GPUfs memory allocator was based on a free list implementation, and required a global lock for each allocation.

Solution: a ring-buffer allocator

Since our system only deals with fixed size allocations of pages, we implement our memory allocator based on the ring buffer data structure. Each cell in the buffer points to a pre allocated page on the GPU memory. Allocating new pages can be implemented by a simple atomic operation, causing minimal contention on the memory allocator.

The ring-buffer memory allocator reduces the overall buffer cache access overhead by 9× even when dealing with the maximum request load of 200K requests per second that can be delivered by the system.

In conclusion, the end-to-end performance of the complete buffer cache implementation with the new memory allocator and concurrent hash table is presented in Table 3.

Image size	Total search time
512x384	3.86ms (22.20%)
1024x768	15.96ms (25.30%)
2048x1536	130.32ms (47.47%)
4096x3072	345.87ms (34.72%)

Table 3: Final hash table search time. The number in parentheses is the percentage of the total running time (Not including IPC transfers)

Figure 5: Effective bandwidth as a function of transfer buffer size

5.3 Optimizing CPU-GPU transfers

Another known bottleneck in many GPU systems is the memory transfer bandwidth between the CPU and the GPU over the PCIe bus. This bandwidth is particularly critical in I/O-bound LSH-based applications. We measure the effective bandwidth of the memory transfers as the ratio between the total amount of data sent from the CPU to the GPU and the total transfer time. Our results show the average bandwidth of 0.5GB/s across all the input images, with the original GPUfs performing 4KB transactions. In fact, as we see from a microbenchmark that evaluates data transfer bandwidth in Figure 5, the performance of 4KB transfers is limited to 1.4GB/s, which is only 13% of the 11GB/s achievable for 1MB transfers.

The original GPUfs system was not designed to use small page sizes, and produced the best performance with much larger pages of 128KB and 256KB per page. Using such large pages in the image collage application, however, is extremely inefficient. This is because the data accesses do not experience spatial locality, so 124KB out of a 128KB page transfer would never be used, and would waste both PCI bandwidth and GPU memory used for the buffer cache.

Solution: I/O Batching

We introduce batching in the Remote Procedure Call mechanism of GPUfs, which is responsible for issuing actual file system requests from GPU to the internal file server running on a CPU.

The change involves two parts: batching of requests, and dispatching of responses to the respective pages in the buffer cache.

The first part is carried by a CPU. We apply batching to the requests in the RPC request queue, which is used by the GPU to enqueue file access requests, and is shared between

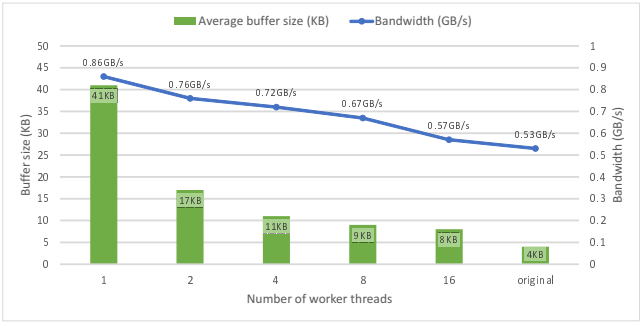

Figure 6: Effective bandwidth

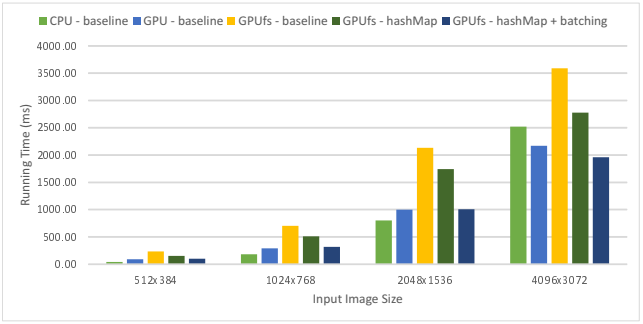

Figure 7: Image collage performance using the optimized version of GPUfs.

the CPU and the GPU. The CPU worker thread periodically traverses the queue and gathers pending requests. For each pending request, the thread reads the requested page from the disk and stores it in a designated staging buffer in CPU pinned memory, eventually building a set of multiple pages to be transfered into GPU memory. When no more requests are pending, or the buffer is full, the data is transferred into a staging area in GPU memory.

The second part is performed on the GPU. After the data transfer is complete, the GPU thread that issued a request copies the data from the staging area into its final location, i.e. in the respective GPU buffer cache page. Note that this memory copy is not necessary in the original GPUfs design because the file data from CPU is transferred directly into the page in the GPU buffer cache. This design is efficient for handling only large pages, however.

We evaluate the batching mechanism in the context of the collage application using the largest image and show the results in Figure 6. We use a different number of CPU worker threads to speedup the RPC queue processing. We compare both the average transfer size and the effective bandwidth. Observe that somewhat counter-intuitively, increasing the number of worker threads reduces the overall memory bandwidth. This is because the average transfer size drops significantly for more threads. This workload achieves a maximum effective bandwidth of 0.86GB/s, which represents only a modest improvement over the original GPUfs. We intend to further investigate dynamic batching mechanisms to achieve higher performance in the future.

6. RESULTS

We combine all the optimizations and show the end-to-end application performance in Figure 7. The new GPUfs implementation achieves 2× speedup on average over the original GPUfs on the image collage application. Importantly, our optimizations of the GPUfs layer allow the GPUfs-based collage implementation to outperform the 12-core CPU implementations by 30% and the native CUDA-based GPU implementation by 11% while using 20% fewer lines of code.

7. REFERENCES

[1] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: Gpu-accelerated incremental storage and computation. In *FAST*, page 14, 2012.

[2] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 211–224. ACM, 2013.

[3] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[4] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer, 2006.

[5] F. Ji, H. Lin, and X. Ma. Rsvm: a region-based software virtual memory for gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 269–278. IEEE, 2013.

[6] J. Lee, M. Samadi, and S. Mahlke. Vast: the illusion of a large memory space for gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 443–454. ACM, 2014.

[7] J. Pan and D. Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 211–220. ACM, 2011.

[8] S. Petrović, M. Osborne, and V. Lavrenko. Streaming first story detection with application to twitter. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 181–189. Association for Computational Linguistics, 2010.

[9] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[10] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating a file system with gpus. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 485–498. ACM, 2013.

[11] A. Stupar, S. Michel, and R. Schenkel. Rankreduce–processing k-nearest neighbor queries on top of mapreduce. In *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval*, pages 13–18. Citeseer, 2010.