Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives

Michael Gowanlock
Northern Arizona University
School of Informatics, Computing, and Cyber Systems
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

Ben Karsin
Université libre de Bruxelles
Department of Computer Science
Brussels, Belgium
Benjamin.Karsin@ulb.ac.be

Zane Fink
Northern Arizona University
School of Informatics, Computing, and Cyber Systems
Flagstaff, AZ, U.S.A.
zwf5@nau.edu

Jordan Wright
Northern Arizona University
School of Informatics, Computing, and Cyber Systems
Flagstaff, AZ, U.S.A.
jaw566@nau.edu

ABSTRACT

Many database operations have a low compute to memory access ratio. In heterogeneous systems, where a graphics processing unit (GPU) is interconnected via PCIe, the data transfer bottleneck is perceived as insurmountable to achieving performance gains on these memory-bound database primitives. On the other hand, several compute-bound database operations have been shown to achieve significant performance gains using the GPU. This leads to CPU-only memory-bound applications having an increasingly non-negligible impact on database query throughput. In this paper we examine several of these overlooked algorithms, including (*i*) batched predecessor searches; (*ii*) multiway merging; and, (*iii*) partitioning. We examine the performance of parallel CPU-only, GPU-only, and hybrid CPU/GPU approaches, and show that hybrid algorithms achieve respectable performance gains. We develop a model that considers main memory accesses and PCIe data transfers, which are two major bottlenecks for hybrid CPU/GPU algorithms. The model lets us analytically determine how to distribute work between the CPU and GPU to maximize resource utilization while minimizing load imbalance. We show that our model can accurately predict the fraction of work to be sent to each architecture, and consequently, confirms that these overlooked database primitives can be accelerated despite their memory-bound nature.

CCS CONCEPTS

• **Information systems** → *Data management systems*; • **Computing methodologies** → *Parallel algorithms*; • **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

GPGPU, Heterogeneous Systems, Hybrid Algorithms, In-memory Database, Memory-Bound Algorithms

ACM Reference Format:
Michael Gowanlock, Ben Karsin, Zane Fink, and Jordan Wright. 2019. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3329785.3329926

1 INTRODUCTION

Many compute-intensive database operations have been parallelized using new hardware such as graphics processing units (GPUs). Such operations include index searches [23, 30, 35], join operations [4, 27], and range queries [24]. While compute-intensive operations have seen performance gains using GPUs, many database primitives have not been accelerated due to their perceived work-efficiency. Typically, such algorithms perform many operations in-memory and have a low compute to memory access ratio. These algorithms are generally considered to be memory-bound and are not considered as candidates for acceleration. However, as more computationally intensive database operations become less expensive through the exploitation of massively parallel architectures, some of these overlooked algorithms begin to have a non-negligible impact on database query throughput.

One approach to improve the performance of this class of memory-bound algorithms is to develop hybrid parallel algorithms that use both CPU and GPU resources, where each architecture performs part of the total computation. Several algorithms have been designed for parallel computation on multi-core CPUs, which split the work between processing elements. However, most GPU research is dedicated to GPU-only approaches which solve the entire computation on the GPU. This is a missed opportunity, as both the CPU and GPU can be used to compute these database primitives.

In current heterogeneous CPU/GPU systems, the GPU global memory bandwidth is an order of magnitude higher than the CPU-GPU interconnect (e.g., PCIe v3.0 has 32 GiB/s bidirectional bandwidth [33] and Nvidia Volta has 900 GiB/s global memory bandwidth [32]). Thus, for data-intensive memory-bound algorithms,

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329926

the CPU-GPU interconnect is the performance bottleneck. However, if this bottleneck can be overcome, there is an opportunity to exploit the GPU's high memory bandwidth.

In this paper, we propose *accelerating the unacceleratable* — which we define as memory-bound database primitives that are well-suited to a hybrid CPU/GPU execution but not necessarily a GPU-only execution. As a demonstration of the potential improvement over CPU-only primitives, we develop hybrid CPU/GPU algorithms to efficiently solve the following problems: (*i*) batched predecessor searches; (*ii*) multiway merging; and, (*iii*) k-way partitioning.

These three database primitives are used in several database applications. For instance, partitioning and multiway merging are used in several sorting algorithms which are used in "distinct" and "order by" SQL queries [38]. Multiway merging is useful in other contexts, such as indexing [17], and partitioning is used in many database approximation problems [20]. Predecessor searches are frequently used as part of index searches [39].

Algorithms that solve the three database primitives on multi-core CPUs are dominated by main memory accesses. And, as described above, the primary bottleneck in GPU algorithms are CPU-GPU data transfers. Thus, we focus on main memory accesses when modeling both CPU and GPU performance. Since both the CPU and GPU access main memory in "blocks" and have internal caches, we use the well-known external memory (EM) model [1] (also known as the disk access model (DAM)). To minimize memory accesses, we base our CPU, GPU, and hybrid approaches on algorithms that are known to be optimal in the EM model.

The external memory model considers a fixed internal memory size M, block size B, and accessing a block of B elements (called an I/O) has unit cost. All computation in internal memory is considered free. While this model is well-known, it oversimplifies the performance impacts of modern memory systems. For example, CPUs have multi-level cache hierarchies and use approximate "LRU-like" page replacement policies [28] that are difficult to model. Additionally, GPU memory transfers can be overlapped and rely on pinned memory buffers to achieve peak throughput [15]. Thus, rather than considering a fixed memory size M and block size B, we use the total number of main memory *elements* loaded/stored as our performance metric. We base our approaches on EM-optimal algorithms to minimize main memory accesses by both the CPU and GPU, and use benchmarks and experimental results to further improve performance on our hardware platform. In Table 1 we summarize the algorithms that we consider in this work, along with the total number of elements loaded/stored in main memory, as well as the total work performed, according to the standard RAM model [6]. See Table 2 for descriptions of the parameters used. To simplify our analysis, we omit small additive terms (e.g., $+k$) from the total number of main memory accesses. We note that, in our GPU algorithms, since we overlap data transfers to and from the GPU, the total data accessed is the maximum unidirectional data transferred (not the total elements accessed in main memory).

In the context of hybrid CPU/GPU algorithms for memory-bound database primitives, this paper makes the following contributions:

- We show that the EM model can be used to design efficient in-memory hybrid CPU/GPU memory-bound database primitives.

Table 1: Summary of algorithms considered in this paper, total elements loaded/stored from main memory, and asymptotic time complexity in the RAM model. We omit small additive terms for loads/stores and, for the GPU, we compute it as the maximum unidirectional data transferred, since we overlap data transfers. The parameter μ depends on the hardware which is optimized experimentally.

Algorithm	Arch.	Elements accessed in memory	RAM Complexity
Batched Pred. Search	CPU	$3n$	$O(n)$
Batched Pred. Search	GPU	$2n$	$O(n \log n)$
Multiway Merge	CPU	$2n$	$O(n \log k)$
Multiway Merge	GPU	n	$O(n \log k)$
k-way Partition	CPU	$2n \lceil \log_\mu k \rceil$	$O(n \log k)$
k-way Partition	GPU	n	$O((n + n_b k) \log \frac{n}{n_b})$

- We demonstrate that when the CPU and GPU require the same amount of data transferred into their respective architectures per work unit computed, the best hybrid performance is achieved when the majority of computation is performed by the CPU.
- We find that when an algorithm requires a large memory cache to avoid many random memory accesses, the hybrid algorithm achieves the best performance when the majority of the work is assigned to the GPU.
- We show that our model is very accurate at splitting the work between CPU and GPU architectures.
- Across most experimental scenarios we find that the hybrid CPU/GPU database primitives outperform CPU- and GPU-only approaches.

The paper is organized as follows. Section 2 outlines related work. Section 3 describes the hybrid database primitives and models that we propose. In Section 4 we experimentally demonstrate the effectiveness of the hybrid algorithms, and the utility of the models. Finally, Section 5 concludes the paper.

2 BACKGROUND & RELATED WORK

In what follows, we give an overview of the problem, then we discuss several categories of related work, including GPU modeling studies, the database primitives that we implement in this work, and optimizations related to data transfers.

2.1 Problem Statement

For each of our database primitives we implement CPU-only, GPU-only, and hybrid CPU/GPU algorithms. We consider a platform with multi-core CPUs and a GPU, where the total response time includes all data transfers to and from the GPU and related overheads. The final result of each algorithm is stored in main memory. We assume that each algorithm can exceed the GPU's global memory capacity. However, each algorithm may not exceed main memory capacity, as we do not consider the impact of disk accesses in this work.

2.2 Modeling Studies

There is a substantial body of work modeling the performance of applications on GPUs and general models of GPU computation. For example, Schaa and Kaeli [36] propose a multi-GPU model for predicting application response time. The model considers multi-GPU contention for PCIe bandwidth, network bandwidth, and disk access throughput. Their model achieves high accuracy across six applications. Another model [25] combines ideas from the BSP [40], PRAM [11], and QRQW [13, 14] models to create a general model of GPU computation. Other works model GPU performance vs. power trade-offs [18], which are of particular importance in large distributed-memory clusters that consume significant power.

In contrast to some of these previous efforts, we *do not consider the cost of computation.* Also, we address the question of splitting the work between CPU and GPU architectures, instead of focusing on GPU-only algorithms. To our knowledge, no other works address modeling the work splitting strategy that we propose in this paper.

2.3 Database Primitives Explored in this Work

In general, many studies are “GPU-only”, and only optimize GPU computation, while minimally involving the host (e.g., using the host to perform data transfers, and other operations peripheral to the computation itself). However, there is a growing trend towards using both the CPU and the GPU to maximize resource utilization (see Mittal and Vetter [29] for an overview of CPU/GPU approaches and classifications). Many such CPU/GPU algorithms have significant computational requirements. We depart from these studies, since we consider highly memory-bound algorithms with minimal computation that are not typical candidates for GPU acceleration. To reiterate, our problem requires computing the result and transferring it back to the host. Therefore, many of the advancements in the GPU algorithms we study are not applicable to our scenario, as the output of these algorithms resides on the GPU, and not on the host. Typically these GPU-efficient algorithms are used as a subroutine for other GPU kernels, so hybrid approaches such as ours can use them to maximize overall performance. For completeness, below we discuss advances in the GPU primitives that we consider in this work.

Since predecessor searches are memory bound, few works have considered optimizing them on GPUs [3, 21]. Batched predecessor searches on the GPU were optimized for execution in shared memory by Karsin et al. [21]. This work focuses on avoiding bank conflicts in shared memory and finds that their two algorithms that eliminate or avoid bank conflicts are more efficient than their naive reference implementation. Berney et al. [3] considered the performance of predecessor search in global memory on the GPU by looking at different search tree layouts. However, neither of these works consider the cost of CPU/GPU data transfers, which is the focus of this work.

Merging is a building block of many fundamental algorithms (i.e., sorting), so several previous works have looked at ways of optimizing it on GPU architectures [16, 19, 22]. Green et al. [16] and Hou et al. [19] optimized pairwise mergesort algorithms, and Karsin et al. [22] designed a GPU-efficient multiway mergesort algorithm.

Partitioning, like merging, is a building block of some sorting algorithms (e.g., distribution sort). While it has been extensively studied on the CPU [34], it has only been considered on the GPU in the context of sorting, i.e., it has been used as a subroutine for GPU-efficient distribution sort algorithms [7, 26]. We note that the algorithms that these papers propose were designed for older GPUs and may not perform as well on newer architectures [22].

Many of the efforts in the literature outlined in this section focus on improving the performance of GPU computation. Since the CPU/GPU performance of the algorithms that we consider are bound by data transfers (PCIe bandwidth), we *ignore the cost of computation.* Thus, we do not consider such optimizations in this work.

2.4 Data Transfer Optimizations

The work of Fang et al. [9] studied reducing GPU memory transfer overheads by compressing data on the CPU. Similarly, Funke et al. [12] improve query throughput by using compiler optimizations that merge operations together and reduce bandwidth demand. Since we consider hybrid algorithms, we wish to avoid the additional CPU computation overhead of these techniques, so we do not use them in this work.

Gowanlock and Karsin [15] proposed a heterogeneous sorting algorithm for CPU/GPU architectures. They demonstrated that there are several overlooked bottlenecks in CPU/GPU computation, including pinned memory allocation cost and host-to-host memory copies. We employ some of their strategies for our hybrid algorithms. In particular, we use multiple CUDA streams to overlap PCIe data transfers in each direction with computation on the CPU and GPU. Also, we use small pinned memory buffers to avoid the expensive allocation cost. This is particularly important for the memory-bound algorithms that we study. Because the total execution time is low relative to compute-bound applications, small sources of overhead can have a large impact on performance.

3 HYBRID ALGORITHMS

We use the notation in Table 2 when describing our algorithms.

Table 2: Summary of notation and descriptions.

	Description
n	Input size.
p	The number of CPU cores.
α	Measured read/write bandwidth between CPU and main memory for the CPU component of a hybrid algorithm.
β	Measured unidirectional bandwidth between the GPU's global memory and main memory over PCIe.
μ	Number of partition buckets in a single pass. The parameter is a function of the memory system and we optimize it experimentally.
n_b	The number of batches. The total work is divided into these disjoint independent workloads.
HtoD	A data transfer from host to device.
DtoH	A data transfer from device to host.

3.1 Saturated Bandwidth Assumption

As discussed in Section 1, we use the external memory model to ensure that our algorithms minimize the number main memory accesses. For the CPU and GPU, this translates to minimizing the total number of main memory elements loaded or stored. For the GPU, this corresponds to the total amount of data transferred from main memory to global memory. However, data transfer time depends on platform-specific characteristics regarding memory bandwidth and data transfer efficiency. Thus, to avoid over-complicating the model, we assume that all data transfers saturate memory bandwidth and achieve peak throughput. On the CPU, this corresponds to the peak bandwidth of main memory and for the GPU this is the peak PCIe bandwidth.

Since the algorithms we consider are memory bound and easily parallelizable, we are able to saturate both main memory bandwidth and PCIe bandwidth. We saturate main memory bandwidth using the CPU by employing multiple threads for reading/writing data. On the GPU, we use several CUDA streams to saturate PCIe bandwidth, where CPU threads orchestrate memory transfers between the host and GPU. We also use the pinned memory data transfer techniques of Gowanlock and Karsin [15] to maximize CPU-GPU data transfer performance.

3.2 Optimality Assumption

In the CPU-only, GPU-only, and hybrid variants of the algorithms that we describe in Table 1, all of the algorithms are optimal in the external memory model. We transfer the minimum amount of data between main memory and the respective architectures. This assumption guides the design of I/O efficient database primitives.

3.3 Hybrid Primitives Using Batches

The three algorithms that we consider are parallelizable across architectures while minimizing memory accesses. We accomplish this by breaking up the total work into several *batches* of divisible workloads that can be computed independently on either architecture. We define n_b to be the number of batches. We set n_b to be sufficiently high such that the total work for each batch is low to avoid the effects of load imbalance at the end of the computation. For batched predecessor search and partitioning, we arbitrarily select $n_b = 400$, while for multiway merge, we make n_b a function of k to ensure data transfers are sufficiently large to mitigate overheads. The resulting batch size allows us to execute batches that fit within global memory while the total memory footprint may exceed global memory capacity. However, the total memory footprint does not exceed main memory capacity, as all processing occurs in-memory.

3.4 Batched Predecessor Search

We outline the batched predecessor search as follows. Let A be a set of keys sorted in non-decreasing order, where each key is denoted as a_i, where $i = 1, 2, \ldots, n$, and B be a set of queries sorted in non-decreasing order, where each query is denoted as b_j, where $j = 1, 2, \ldots, n$. For each query, $b_j \in B$, the batched predecessor search finds the largest value of i, such that $a_i \leq b_j$. While A and B can vary in size, for simplicity, we assume $|A| = |B| = n$.

We focus on the *batched* predecessor search because it can be used as a subroutine in database operations. Additionally, computing a single query will be unable to saturate GPU resources.

3.4.1 *GPU Algorithm.*

To exploit the GPU's massive parallelism, it is crucial that each query, b_j, be independent of all other queries. Thus, to perform the batch predecessor search on the GPU, we execute the Thrust library's [2] upper bound binary search, which requires $O(n\log n)$ work. Recall that we assume the GPU work is ignored. Consequently, we only consider data transfer to/from the GPU. The batched predecessor search requires sending a total of $2n$ elements to the GPU (*HtoD*), and sending n elements from the GPU back to the host (*DtoH*). Considering only the data transferred, the total data transferred with constant factors is $2n + n = 3n$.

Our model assumes that all data transfers are overlapped between *HtoD* and *DtoH*, thus fully exploiting bidirectional PCIe memory bandwidth. While in practice not all *HtoD* and *DtoH* data transfers will be overlapped, this assumption allows us to negate some of the data transfer overhead in our model. If we let s and r be the total number of *HtoD* and *DtoH* elements transferred, respectively, then the total data we consider transferred is $\max(s, r)$. For the batch predecessor search, the data transferred is therefore $\max(2n, n) = 2n$.

3.4.2 *CPU Algorithm.*

In contrast to the GPU algorithm, the parallel CPU algorithm can take advantage of the batched execution. Instead of performing binary searches, the CPU algorithm executes a *merge find* (finding the index without merging) for each $b_j \in B$. Merge find has been used in other algorithms, such as set intersection [8]. The algorithm is trivially parallelized by assigning n_b batches of sublists of A and B to each processor, where each processor performs a scan over several batches of size n/n_b elements to find the predecessor of each query. Thus, the total work is $O(n)$ across all p processors.

We read a total of $2n$ elements from main memory into the CPU, and write n elements back to main memory. Therefore, the total data transferred with constant factors is: $2n + n = 3n$.

3.4.3 *Hybrid Algorithm.*

To combine the GPU and CPU algorithms, we split the work between each architecture, where we assign a fraction of the n queries to the CPU and GPU. As noted in Section 3.3, we split the work into a number of divisible workloads called *batches*, which allows us to assign work to the CPU or GPU. For the batched predecessor search, we split A and B into n_b value disjoint batches based on the values in each. This ensures that all batches can be computed independently on either architecture. Each batch is denoted as B_i, where $i = 1, 2, \ldots, n_b$. Figure 1 shows an illustrative example of splitting A and B into disjoint batches. Since A and B are sorted, we find the pivots in B as a function of the values in A by performing a binary search for the index that splits the data based on a given value $a \in A$, such that we obtain value-disjoint batches. For example, if $x \in B_i$ and $y \in B_{i+1}$, then $x < y$. Because the batches are value-disjoint, each batch contains $\approx \frac{n}{n_b}$ elements from A and B. Since $n_b \ll n$, the time to generate the batches is negligible compared to the time to process the batches.

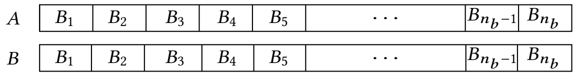


Figure 1: Illustrative example of splitting A and B into value-disjoint batches for the batched predecessor search.

We model evenly splitting the queries based on PCIe and memory bandwidth to obtain low load imbalance (i.e., architectures finish computing their respective queries at similar times). Let β be the unidirectional bandwidth over PCIe, and α be the memory bandwidth between the CPU and main memory when simultaneously reading and writing, where β and α are given in elements per second. For a given platform, β and α can be obtained through simple microbenchmarks.

The total time to execute the CPU- and GPU-only algorithms are denoted as T^{CPU} and T^{GPU}, respectively, and we estimate them to be:

$$T^{CPU} = \frac{3n}{\alpha} \text{ and } T^{GPU} = \frac{2n}{\beta}. \tag{1}$$

To split the work between architectures, let f be the fraction of n elements computed on the CPU, where $1 - f$ is the fraction of n computed on the GPU. Using the total work in our model (Equation 1) and substituting f for n, let

$$T^{CPU} = \frac{3f}{\alpha}, \text{ and } T^{GPU} = \frac{2(1-f)}{\beta}. \tag{2}$$

We compute f as a function of the parameters α and β, and set $T^{CPU} = T^{GPU}$, such that we model each architecture completing its computation at the same time. Therefore,

$$\frac{3f}{\alpha} = \frac{2(1-f)}{\beta} \tag{3}$$

$$f = \frac{2\alpha}{2\alpha + 3\beta}. \tag{4}$$

Thus, given the constants α and β, we obtain the fraction of the total queries that should be executed on the CPU and GPU to minimize load imbalance.

3.5 Multiway Merging

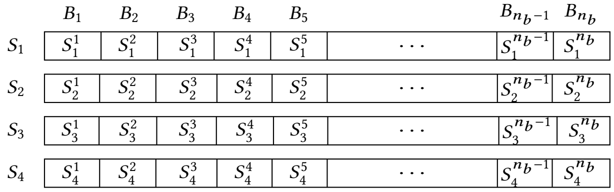
We define the problem of *multiway merging* as follows. Given input array A consisting of k sublists, denoted as S_j, where $j = 1, 2, \ldots, k$, each of size $\frac{n}{k}$ and sorted in non-decreasing order, we wish to output the n total elements in sorted order. Furthermore, we assume that k is small enough that we can load elements from each sublist into memory without degrading CPU cache utilization. In the CPU-only, GPU-only and hybrid algorithms described below, all are optimal in the external memory model, as we transfer the minimum amount of data between main memory and the respective architecture.

3.5.1 GPU Algorithm. Our GPU multiway merge algorithm begins by dividing A into n_b batches that contain elements from all k lists that form an interval of the sorted result. That is, batch B_i contains roughly $\frac{n}{n_b}$ elements from A and, if $x \in B_i$ and $y \in B_{i+1}$, then $x < y$, for any $1 \le i \le n_b$. We define S_j^i to be the subset of the sublist S_j that is in batch B_i, so $B_i = \bigcup_{j=1}^{k} S_j^i$. To divide A into these batches that are value-disjoint and of roughly equal size, we find pivots that define the range of values contained in each batch. We consider small values of k in this work to find pivots that divide A into our n_b batches with minimal overhead.

Using a number of CUDA streams (and pinned memory to achieve peak throughput), we transfer batches to the GPU and merge each batch into a sorted list. We perform this merging on *each batch* by repeatedly using the Thrust library's [2] pairwise merge (e.g., pairwise merging $k - 1$ times). Sorted batches are then transferred back to main memory. We form the final output by simply concatenating the sorted batches together (i.e., $B_1 B_2 \ldots B_{n_b}$).

Figure 2 shows an illustration of splitting the sorted sublists S_j, with $k = 4$ sublists. Each batch is processed by a single CUDA stream, but multiple streams are used for merging the batches, B_i.

Note that since we assume all bidirectional data transfers are overlapped in CUDA streams to/from the GPU, our model estimates that a total of n elements are transferred between main memory and the GPU.



	B_1	B_2	B_3	B_4	B_5		B_{n_b-1}	B_{n_b}
S_1	S_1^1	S_1^2	S_1^3	S_1^4	S_1^5	$\cdots$	$S_1^{n_b-1}$	$S_1^{n_b}$
S_2	S_2^1	S_2^2	S_2^3	S_2^4	S_2^5	$\cdots$	$S_2^{n_b-1}$	$S_2^{n_b}$
S_3	S_3^1	S_3^2	S_3^3	S_3^4	S_3^5	$\cdots$	$S_3^{n_b-1}$	$S_3^{n_b}$
S_4	S_4^1	S_4^2	S_4^3	S_4^4	S_4^5	$\cdots$	$S_4^{n_b-1}$	$S_4^{n_b}$

Figure 2: Illustrative example of splitting $k = 4$ sorted sublists to be transferred to the GPU to be merged. Each individual batch B_i is merged using a single CUDA stream.

3.5.2 CPU Algorithm. The CPU algorithm simply uses the GNU parallel mode extensions [37] to perform a multiway merge. The algorithm reads a total of n elements into the CPU and writes a total of n elements back to main memory. Thus, the total number of elements accessed by this algorithm is $2n$.

3.5.3 Hybrid Algorithm. As with the predecessor search, we combine the GPU and CPU algorithms by splitting the work between each architecture, where we assign a fraction of the n_b batches to the CPU and the GPU. We use the same method of computing f as we did for predecessor search in Section 3.4.3.

$$T^{CPU} = \frac{2n}{\alpha} \text{ and } T^{GPU} = \frac{n}{\beta}. \tag{5}$$

Thus we compute f to be:

$$f = \frac{\alpha}{\alpha + 2\beta}. \tag{6}$$

3.6 Partitioning

We consider the problem of *k-way partitioning* (or simply partitioning). Given an unsorted list, A, of n elements, we wish to partition A into k buckets $A_1, A_2, \ldots, A_k$ of roughly equal size such that each bucket is value-disjoint. That is, for any two elements $a \in A_i$ and $b \in A_j$, if $i < j$, then $a < b$. This problem is also known as *distribution* and is a subroutine of many algorithms, including distribution sort (also known as multiway quicksort). In the RAM and EM models, the lower bounds for partitioning n elements into k buckets is $O(n \log k)$ and $O(\frac{n}{B} \log_{M/B} k)$, respectively. The external memory bound can be achieved by repeatedly partitioning n into $\frac{M}{B}$ buckets (which can be done in a single I/O-efficient scan). We note that if $k = n$, partitioning is equivalent to sorting.

Partitioning involves (i) finding pivots for each bucket (k total); (ii) determining which bucket each element is in; and, (iii) moving each element into contiguous memory with other elements in the same bucket. Since we focus on the memory-bound "bucketing" portion of the problem, we assume that the pivots are given and no more than a constant factor more than $\frac{n}{k}$ elements will end up in any bucket. The I/O-efficient approach for partitioning reads the data while maintaining a local cache for each bucket. When one becomes full, it is written to the bucket in main memory. Thus, in the external memory model, a cache of a size B is needed for each bucket, so it can be partitioned into $\frac{M}{B}$ buckets during a single scan. If $k > \frac{M}{B}$, multiple scans are required, so n elements can be partitioned in $\frac{n}{B}\lceil\log_{M/B} k\rceil$ I/Os. Since we do not use a fixed M and B, we define μ to be the number of buckets partitioned at each pass over the data (i.e., the $\frac{M}{B}$ in the I/O model). Figure 3 illustrates how we can partition an unsorted input into k buckets by partitioning into μ buckets for $\log_{\mu} k$ rounds.

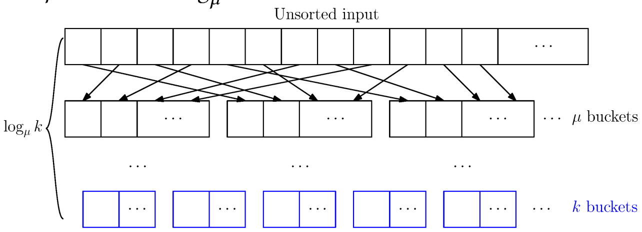


Figure 3: Illustration of partitioning an input into k buckets by repeatedly partitioning into μ buckets, as is performed by our CPU-only algorithm.

3.6.1 GPU Algorithm. Recall that, in the external memory model, we require $\lceil\log_{M/B} k\rceil$ rounds to partition into k buckets. Rather than fixed memory and block size, we use a single parameter, μ, that is based on the amount of data that can be loaded into internal memory without losing performance. For the GPU, we know that internal memory (global memory) is large in relation to n. For example, on our hardware platform, main memory has a maximum capacity of 128 GiB, while our GPU has 16 GiB of global memory. Thus, we can conclude that μ is large and, for reasonable values of k, we can partition into k buckets in a single round.

With the large internal memory, the high computational throughput, and efficient libraries available on the GPU, we further simplify our k-way partitioning algorithm by *sorting* batches internally rather than bucketing elements. Thus, our GPU k-way partitioning algorithm operates as follows. We first transfer the k pivots to the GPU. We then divide the input into n_b equal batches and sort each batch on the GPU. After sorting each batch, the GPU performs binary search to determine which portions of the batch belong to each bucket and transfers this information back to the CPU. Finally, when copying the batch from global memory back into main memory, the data is simply placed in the correct bucket.

Using this algorithm, we simply transfer the pivots and data in each batch to the GPU once and then send the result back to the host. Since data transfers are overlapped and we ignore small additive constants, we say that the GPU accesses n elements. We note that the GPU internally performs much more work (i.e., $O(n \log \frac{n}{n_b} + n_b k \log \frac{n}{n_b})$ work in the RAM model), but we do not consider internal work in our analysis.

3.6.2 CPU Algorithm. Unlike the GPU, the CPU has a much smaller cache size, and, although we do not used a fixed M due to the multilevel cache, we know that μ is much smaller for the CPU than the GPU. Therefore, we cannot assume that $k \leq \mu$ and we must use multiple passes to partition for large values of k. At the first pass, we partition the input into μ buckets. At the next pass, we further partition each bucket into μ buckets (for μ^2 partitions), and so on (illustrated in Figure 3).

Our CPU implementation uses multiple threads, so each thread is assigned a subset of the input and maintains a cache for each bucket. The input is scanned and, as caches become full, threads write them to a shared output in main memory. At each pass, each thread maintains μ small caches (we use 1024 elements per cache). We assume that the k pivots of each round fit in cache as well and are read at the beginning. Thus, during each pass, each of the p threads reads $\frac{n}{p}$ elements and writes $\frac{n}{p}$ elements back, for a total of $2n$ elements among all threads. Our CPU k-way partitioning algorithm loads and stores a total of $(2n + k)\lceil\log_{\mu} k\rceil$ elements in main memory (omitting small additive terms). We experimentally investigate the performance impact of μ and determine the best value for our platform in Section 4.

3.6.3 Hybrid Algorithm. The hybrid approach to partition is quite simple since the input data does not need to be preprocessed at all. We simply divide our initial input into n_b batches and assign a subset of batches to the GPU and a subset to the CPU. We then perform the above algorithms on the assigned batches. Once both the CPU and GPU finish processing their batches, we simply combine the two sets of buckets. We note that we can actually combine the sets of buckets while we are transferring the data back from the GPU so that there is minimal additional overhead.

To determine how much work to assign to the CPU versus the GPU, we first estimate the time to partition on the CPU and GPU (similarly to Section 3.4.3):

$$T^{CPU} = \frac{(2n)\lceil\log_{\mu} k\rceil}{\alpha} \text{ and } T^{GPU} = \frac{n}{\beta}. \tag{7}$$

Thus, the fraction of work that we assign to the CPU is

$$f = \frac{\alpha}{\alpha + (2\beta\lceil\log_{\mu} k\rceil)}. \tag{8}$$

Note that the number of passes required by the CPU algorithm (computed by μ and k) dictates the work distribution between the CPU and GPU, in addition to the main memory and PCIe bandwidth.

4 EVALUATION

4.1 Experimental Methodology

Our platform contains 2× Intel Xeon E5-2620 v4 processors, with 16 total physical cores, at a clock rate of 2.1 GHz, and 128 GiB of main memory, equipped with a Quadro GP100 with 16 GiB of global memory. The host code is compiled with the GNU compiler with the O3 optimization flag and parallelized using OpenMP [5], and the GPU code is written in CUDA 9 [31]. All results are averaged over 5 trials, and all algorithms use 64-bit data elements. Since all algorithms are designed to use divisible workloads that are executed in batches, across all batched predecessor search and partitioning experiments, we select $n_b = 400$ batches to be executed. For multiway merge, we use a variable batch size based on k to ensure that data transfers are sufficiently large to achieve peak throughput.

We demonstrate the performance of CPU-only, GPU-only, and hybrid primitives (for each of the algorithms we study in Table 1). Their configurations are described as follows.

- **CPU-only:** The CPU-only algorithms are executed with 16 threads (the number of physical cores on the platform).
- **GPU-only:** The GPU-only algorithms are executed using 8 streams (with 8 CPU threads) to saturate memory bandwidth for *HtoD* and *DtoH* data transfers. Also, each stream uses pinned memory buffers to incrementally copy the data in either direction (*HtoD* or *DtoH*) for each of the arrays described in the respective algorithms. Each pinned memory buffer is of size 8 MiB, which is reused for data transfers in each stream. Since the pinned memory buffers are small, we obviate expensive allocation costs. See [15] for an overview of pinned memory allocation costs.
- **Hybrid:** The hybrid algorithm simply combines the CPU-only and GPU-only algorithms above, using a total of 24 threads.

In many algorithms that focus on improving the performance of the algorithm itself (e.g., on-GPU sorting [22]), it is customary to evaluate the sensitivity of the algorithm to different input distributions. Examples in the context of sorting include: high frequency of duplicates in a list, exponentially distributed input lists, or the list is already sorted. Since data transfers dominate response time, and we exclude all computation in our models of each algorithm, we do not perform any such data distribution sensitivity studies.

4.2 Microbenchmarks

To obtain α, we execute a simple microbenchmark which simultaneously reads and writes 5 GiB of 64-bit integers stored in main memory using 16 threads to saturate memory bandwidth. We obtain $\alpha = 19.56$ GiB/s on our platform. Figure 4 shows the scalability of the microbenchmark. For comparison purposes, we plot the read only memory bandwidth (reading 10 GiB of 64-bit integers) and observe that the memory bandwidth is significantly higher than when performing both reading and writing. This is likely due to prefetching, which is not possible when writing to main memory.

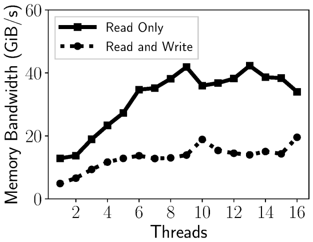


Figure 4: Main memory bandwidth vs. the number of threads. Multiple threads are needed to saturate memory bandwidth. Simultaneous reading and writing has lower throughput than reading only.

To obtain β, we transfer 10 GiB from pinned memory on the host to global memory on the GPU. We use the profiler to measure the bandwidth, and we obtain $\beta = 11$ GiB/s.

4.3 Batched Predecessor Search

Using our model, which splits the queries between the CPU and GPU for the batched predecessor search (Equation 4), and the values of α and β from microbenchmarks, we compute $f = 0.54$ (the fraction of queries sent to the CPU). Figure 5(a) plots the response time vs. the input size of the search, query and result set arrays (n), where the total memory footprint is $3n$. The execution time for both CPU-only and GPU-only algorithms are shown, which demonstrates that the CPU is more efficient than the GPU. The hybrid approach that splits the queries between CPU and GPU is more efficient than CPU-only. For instance, when $n = 5 \times 10^9$, the speedup of the hybrid approach over CPU-only is 1.18×, which is respectable, given that the GPU is limited by both PCIe bandwidth and is typically unsuitable for memory-bound database primitives.

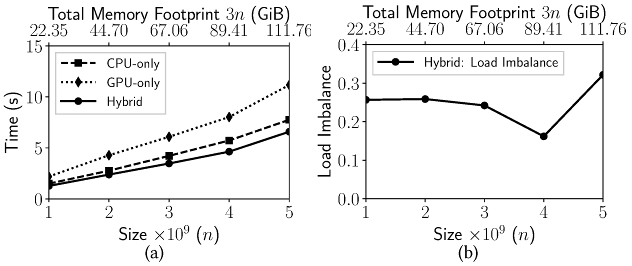


Figure 5: (a) Response time vs. input size (n) comparing CPU-only, GPU-only, and hybrid batched predecessor search algorithms, where the total memory footprint, $3n$, is plotted in GiB on the top horizontal axis. (b) The load imbalance of the hybrid algorithm in (a).

Figure 5(b) plots the fraction load imbalance for the hybrid algorithm[1], which illustrates that the model does not accurately predict an even split for the batched predecessor search. We find that the fraction load imbalance is between 0.16 and 0.32. This large load imbalance is due to our assumption that all data transfers are overlapped. For this algorithm, we find that there are time periods where *HtoD* and *DtoH* data transfers do not occur concurrently. This degrades the GPU's expected throughput.

4.4 Multiway Merge

From Equation 6, we obtain $f = 0.47$ for the fraction of queries sent to the CPU. We evaluate the performance of the model for different values of k where $k \in \{2, 8, 32\}$, where the size of each sublist $S_i = \frac{n}{k}$, $i = 1, 2, \ldots, k$. Note that due to the amount of memory required to perform multiway merging on the device, we set $n_b = \frac{3200}{k}$.[2] Figures 6(a), (c), and (e) plot algorithm response time vs. input size when $k = 2, 8$, and 32 respectively, and similarly Figures 6(b), (d), and (f) plot hybrid load imbalance vs. n.

For $k = 2$ we see the CPU-only algorithm outperforms both the GPU-only and hybrid algorithms for $n < 8.0 \times 10^9$ (Figure 6(a)). This results in an average slowdown of 0.77× for the hybrid algorithm. When $n = 8.0 \times 10^9$, we observe a speedup of 1.03×, and thus predict that for $n \geq 8.0 \times 10^9$ the hybrid algorithm will provide a speedup

[1] Load imbalance is computed as: $|T^{CPU} - T^{GPU}|/T$, where T^{CPU} and T^{GPU} are the times when the CPU and GPU finish executing their batches, respectively, and T is the total response time.

[2] Because we do pairwise merging, and in order to fulfill the assumption of overlapping data transfers, we allocate memory on the device for $2 \cdot k \cdot \frac{n}{n_b} \cdot n_s$ 64 bit integers, where n_s is the number of CUDA streams. Therefore, we cannot use a constant value of n_b across all values of k.

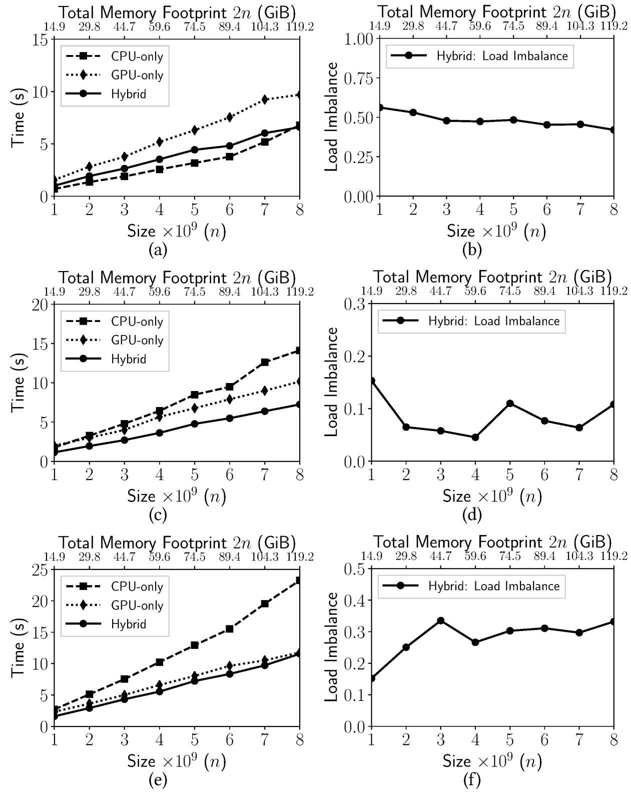


Figure 6: (a), (c), and (e) Response time vs. input size (n) when $k = 2, 8, 32$ respectively, comparing CPU-only, GPU-only, and hybrid multiway merge algorithms, where the total memory footprint, $2n$, is plotted in GiB on the top horizontal axis. (b), (d), and (f) The load imbalance of the hybrid algorithm in (a), (c), and (e).

over the CPU-only algorithm. In Figure 6(b), we see load imbalances between 0.42 and 0.56, with an average of 0.48. In Section 4.6 we elaborate on the cause of the load imbalance in the multiway merge algorithm.

When $k = 8$, we find that the GPU outperforms the CPU for each value of n. We observe in Figure 6(c) the effectiveness of the hybrid algorithm, which achieves a speedup over both the CPU-only and GPU-only algorithms. For example, when $n = 8 \times 10^9$, the speedup is 1.40× over the GPU-only approach, and 1.95× over the CPU-only algorithm. Overall, an average speedup of 1.50× is observed over the GPU-only algorithm. Figure 6(d) plots the load imbalance vs. n. We find load imbalance between 0.04 and 0.15, with an average of 0.09. This indicates that the value of f selected by the model provides good load balancing and results in good performance for multiway merging when $k = 8$.

In the case where $k = 32$, we again see the hybrid algorithm resulting in speedup over the CPU-only and GPU-only algorithms. An average speedup of 1.83× is obtained over the CPU-only algorithm, while an average speedup of 1.17× is observed over the GPU-only algorithm (Figure 6(e)). However, we see further degradation of CPU-only performance with growing values of n, and thus the performance of the hybrid algorithm is reduced. Consider the case where $n = 8.0 \times 10^9$. Here a speedup of 1.02× is achieved over the GPU-only algorithm, and we expect that for $n > 8.0 \times 10^9$ the GPU-only algorithm may outperform the hybrid algorithm. Figure 6(f) displays an average load imbalance of 0.28, demonstrating that the model does not accurately predict a good value of f for $k = 32$.

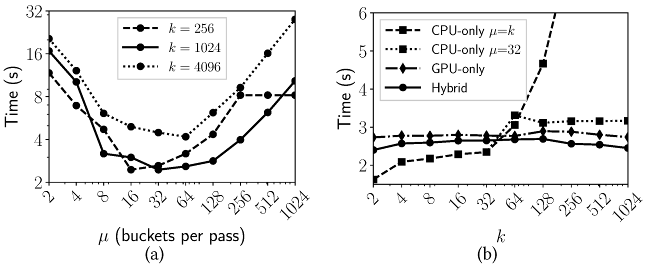


Figure 7: (a) Average response time vs. μ for several k values when $n = 10^9$. (b) Average response time vs. k for CPU-only, GPU-only, and hybrid algorithms where $n = 10^9$.

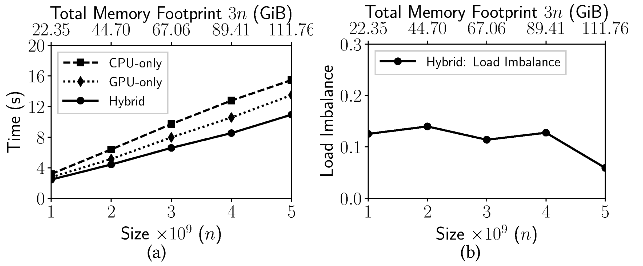
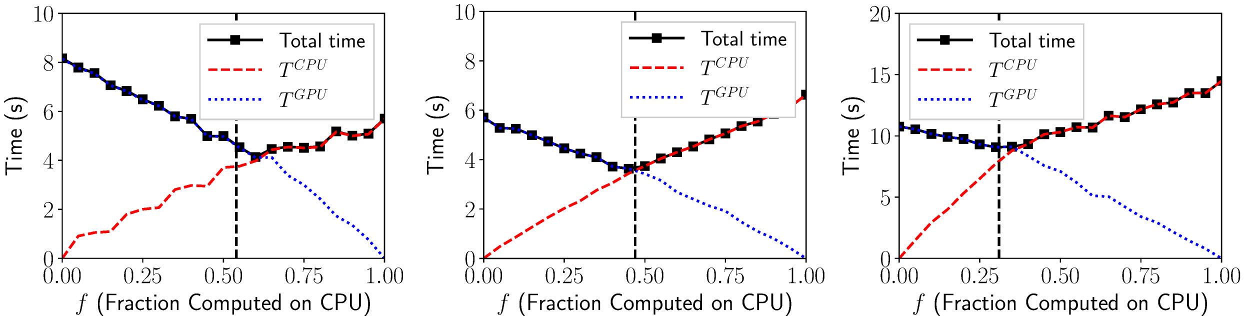


Figure 8: (a) Response time vs. input size (n) comparing CPU-only, GPU-only, and hybrid partitioning algorithms for $k = 1024$ buckets, where the total memory footprint, $3n$, is plotted in GiB on the top horizontal axis. (b) The load imbalance of the hybrid algorithm in (a).

4.5 Partitioning

As discussed in Section 3.6.3, our CPU partitioning algorithm relies on the additional parameter, μ, that determines the number of rounds (and therefore amount of work) that the CPU must perform. This in turn effects the fraction of work, f, that is assigned to the CPU by the hybrid algorithm. Figure 7(a) plots the average response time of CPU-only partitioning versus μ, for several values of k and $n = 10^9$. These results indicate that using too small or too large a value of μ significantly degrades performance. This is expected, as using a small value of μ results in extra passes through memory, while too large a value results in more cache misses. Thus, on our platform we select $\mu = 32$ for all experiments.

Recall that the CPU-only and GPU-only algorithms perform different amounts of work based on k and μ. Thus, the number of buckets, k, has a different impact on the performance of each approach. Figure 7(b) plots the average response time of each algorithm for varying k on inputs of $n = 10^9$ elements. When k is



(a) Batched Predecessor Search, model: $f = 0.54$.

(b) Multiway Merge, model: $f = 0.47$.

(c) Partitioning, model: $f = 0.31$.

Figure 9: Hybrid model accuracy for all algorithms. The total response time, T^{CPU}, and T^{GPU} vs. f, are plotted, where T^{CPU} and T^{GPU} are the times when the CPU and GPU finish computing their work. We show $n = 4.0 \times 10^9$ for all algorithms. The vertical dashed line in each plot denotes the modeled value of f. In (b) we use $k = 8$ for multiway merging, and in (c) we perform partitioning with $k = 1024$ and $\mu = 32$.

very small (< 32), the CPU-only algorithm achieves the best performance because only one pass is performed ($\mu \geq k$) with minimal cache misses. However, for large k, using $\mu = k$ (only one pass through memory) results in many more cache misses, degrading performance. Thus, for reasonably large k (i.e., $k > 32$), the hybrid algorithm achieves the best performance.

We consider partitioning into $k = 1024$ buckets in experiments hereafter. Using $\mu = 32$, $k = 1024$, and the α and β values measured in Section 4.2, from Equation 8 we compute $f = 0.307$ for our hybrid algorithm. Figure 8(a) plots the average execution time of the CPU-only, GPU-only, and hybrid (using $f = 0.3$) partitioning algorithms for various input sizes. We see that, across all input sizes, the hybrid algorithm provides the best performance. Figure 8(b) plots the load imbalance of the hybrid partitioning algorithm and indicates that the $f = 0.3$ selected by our model provides good load balancing. While we achieve good load balancing, the overheads of our hybrid approach (e.g., avoiding race conditions when updating buckets) reduces the maximum speedup that we achieve. Nevertheless, our hybrid algorithm achieves an average speedup of 1.19× and 1.43× over the GPU-only and CPU-only algorithms, respectively.

4.6 Comparison of Hybrid Algorithms: Accuracy of Splitting the Work

For each algorithm (batched predecessor search, multiway merge, and partitioning), we used the load imbalance between architectures as a measure of how well the model split the work between the CPU and GPU. Figure 9 plots the total time, T^{CPU}, and T^{GPU} vs. f (the fraction of work computed on the CPU) for each algorithm, where the modeled value of f is shown by the vertical dashed line. For consistency, we show a single value of $n = 4 \times 10^9$ across all algorithms. The times T^{CPU} and T^{GPU} are when the respective architecture finishes computing their batches, thus the total time is roughly max(T^{CPU}, T^{GPU}).

Regarding the batched predecessor search (Figure 9(a)) we observe that the model predicts a very good value for f. The modeled value is $f = 0.54$, and the best value in the plot is $f = 0.60$. However, from Figure 5(b) we find that the load imbalance is 0.16. This shows that even if the model can predict a good value for f, small differences in f can yield significant load imbalance for the batched predecessor search.

Figure 9(b) demonstrates the accuracy of the model's prediction of f for multiway merging when $k = 8$. We observe that the optimal value of f is almost exactly the value predicted by our model. Furthermore, we observed in Figure 6(d) that the model very accurately predicts the value of f when $k = 8$, where the load imbalance is between 0.04 and 0.16. However, we see in Figure 10 that plots the total response time for CPU-only and GPU-only vs. k (where $n = 4 \times 10^9$), that the value of k greatly impacts CPU-only performance, and thus, the accuracy of the model degrades for other values of k. This explains the high load imbalance in Figure 6(b) and (f). We find that the value of k has a non-negligible impact on CPU-only response time, which suggests that multiway merging is not entirely memory-bound, and computation has an impact on performance. Since our model does not account for computation, the model's accuracy degrades for some values of k.

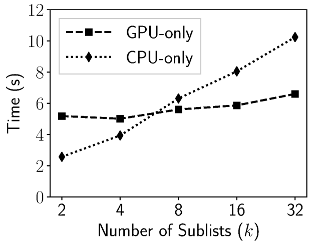


Figure 10: Response time vs. number of sublists (k) for the multiway merge algorithm, where both CPU-only and GPU-only are shown. We show $n = 4 \times 10^9$.

As discussed in Section 3.6, for partitioning our model takes into account the impact of k on CPU performance, as the CPU-only algorithm accesses more data than the GPU-only algorithm. With $k = 1024$ and $\mu = 32$, the CPU version must perform two passes through memory, thus accessing twice as much memory as the GPU. As a result, our model predicts a smaller fraction of work be given to the CPU (i.e., $f = 0.307$). Figure 9(c) indicates that the model prediction is quite accurate, with the best performance achieved when $f = 0.35$ being only slightly faster than the predicted value of 0.3.

Since our model accurately predicts a good value of f, this suggests that we are achieving peak bandwidth utilization between main memory and the CPU, and between main memory and the GPU's global memory over PCIe. If we did not achieve peak bandwidth utilization, then our model would yield an inaccurate value of f.

4.7 Discussion

Our model is able to accurately split work between CPU and GPU architectures. This verifies our assumptions about these memory-bound database algorithms and lets us make several important observations that we discuss as follows.

(1) The external memory model is a powerful starting point for memory-bound algorithms — we can simply ignore the cost of computation and focus on the two performance drivers: main memory accesses and PCIe data transfers.
(2) By focusing on the two performance drivers outlined above, we learn that algorithms with more random memory accesses (i.e., partitioning) are better suited to the GPU, because the internal memory is large and thus, the random memory accesses do not significantly degrade performance. In contrast, the CPU relies on its small multi-level cache to mitigate against slow accesses to main memory; however, random memory accesses reduce the ability of the modern CPU to exploit locality, and the cache becomes ineffective.
(3) The experimental results of Section 4 also indicate that the GPU can substantially improve the performance of algorithms that have sequential data access patterns (i.e., predecessor searches and multiway merging). Sequential memory access patterns are highly efficient on the CPU due to high spatial locality and subsequently, low cache miss rates. However, multi-core CPUs are starved for data due to limited memory bandwidth.
(4) Since CPU performance is limited by main memory bandwidth, increasing the number of CPU cores will not significantly improve performance on the algorithms studied in this paper.
(5) Increasing the CPU-GPU data transfer bandwidth (e.g., with the new NVLink interconnect [32]) will further reduce the response time of the GPU-only algorithm, making hybrid acceleration over a CPU-only approach even more significant.

5 CONCLUSIONS

In this work, we considered three memory-bound database primitives that are typically not candidates for acceleration on GPUs due to the need to perform slow data transfers. However, consider that the measured read/write bandwidth on our platform with 16 threads is $\alpha = 19.56$ GiB/s, whereas the measured unidirectional PCIe bandwidth is $\beta = 11$ GiB/s. The "slow" PCIe interconnect is not significantly slower than the read/write bandwidth between the CPU and main memory. Thus, we can still exploit the GPU for memory-bound database primitives that are often considered unsuitable for GPU acceleration.

Our model only considers main memory accesses and data transfers over PCIe and ignores computation. The model enables assigning the CPU and GPU batches of work such that low load imbalance between architectures is achieved. If an algorithm is better suited to the CPU (i.e., predecessor searches) or the GPU (i.e., multiway merge, partitioning), then the model will assign more work to the respective architecture. At first glance, all of the algorithms studied in this paper are seemingly unsuitable for execution on the GPU. However, our model and experimental results demonstrate that there are significant performance benefits to using a hybrid approach.

New interconnects, such as NVLink [10] or PCIe v.5 [33] will improve the benefits of using GPUs for memory-bound algorithms, by allowing an increase in the fraction of the total work computed on the GPU. For example, our measured PCIe unidirectional bandwidth is 11 GiB/s (69% of the peak bandwidth of 16 GiB/s). If 69% of PCIe v.5 unidirectional bandwidth (64 GiB/s [33]) is achieved, then in our model, $\beta = 44$ GiB/s. Thus, for batched predecessor searches (as an example algorithm), our hybrid algorithm would perform only 23% of the total workload on the CPU. Therefore, if current hardware trends continue, the GPU will become increasingly suitable for a much larger class of algorithms, thus expanding the niche of GPU database operations.

Future work includes investigating whether compression schemes or other memory transfer optimizations can alleviate some of the bottlenecks, despite the computational overhead. Also, this work motivates the study of other fundamental database operations that have not been considered for GPU acceleration.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant OAC-1849559 and Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1.

REFERENCES

[1] Alok Aggarwal and Jeffrey Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.
[2] Nathan Bell and Jared Hoberock. 2012. Thrust: a productivity-oriented library for CUDA. *GPU Computing Gems: Jade Ed.* (2012).
[3] Kyle Berney, Henri Casanova, Alyssa Higuchi, Ben Karsin, and Nodari Sitchinava. 2018. Beyond Binary Search: Parallel In-Place Construction of Implicit Search Tree Layouts. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1070–1079.
[4] Christian Böhm, Robert Noll, Claudia Plant, and Andrew Zherdin. 2009. Index-supported Similarity Join on Graphics Processors. In *BTW*. 57–66.
[5] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.* Morgan Kaufmann.
[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.
[7] Frank Dehne and Hamidreza Zaboli. 2012. Deterministic Sample Sort for GPUs. *Parallel Processing Letters* 22, 3 (2012).
[8] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. 2009. Using Graphics Processors for High Performance IR Query Processing. In *Proceedings of the 18th International Conference on World Wide Web*. 421–430.
[9] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 670–680.
[10] D. Foley and J. Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.
[11] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. 114–118.
[12] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings*

of the 2018 International Conference on Management of Data. ACM, 1603–1618.

[13] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. 1996. The queue-read queue-write asynchronous PRAM model. In *European Conference on Parallel Processing*. Springer, 277–292.

[14] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. 1997. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.* (1997), 638–648.

[15] Michael Gowanlock and Ben Karsin. 2019. A Hybrid CPU/GPU Approach for Optimizing Sorting Throughput. *Parallel Comput.* 85 (2019), 45–55.

[16] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, 331–340.

[17] Steffen Heinz and Justin Zobel. 2003. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology* 54, 8 (2003), 713–729.

[18] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *ACM SIGARCH Computer Architecture News*, Vol. 38. 280–289.

[19] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast Segmented Sort on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, Article 12, 10 pages.

[20] Yannis Ioannidis. 2003. Approximations in database systems. In *International Conference on Database Theory*. Springer, 16–30.

[21] Ben Karsin, Henri Casanova, and Nodari Sitchinava. 2015. Efficient batched predecessor search in shared memory on GPUs. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 335–344.

[22] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. 2018. Analysis-driven Engineering of Comparison-based Sorting Algorithms on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*. 86–95.

[23] Jinwoong Kim, Won-Ki Jeong, and Beomseok Nam. 2015. Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. *IEEE Transactions on Parallel and Distributed Systems* 26, 8 (2015), 2258–2271.

[24] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel and Distrib. Comput.* 73, 8 (2013), 1195–1207.

[25] Kishore Kothapalli, Rishabh Mukherjee, M Suhail Rehman, Suryakant Patidar, PJ Narayanan, and Kannan Srinathan. 2009. A performance prediction model for the CUDA GPGPU platform. In *2009 IEEE International Conference on High Performance Computing*. IEEE, 463–472.

[26] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. 2010. GPU sample sort. In *2010 IEEE International Symposium on Parallel & Distributed Processing*. 1–10.

[27] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A fast similarity join algorithm using graphics processing units. In *2008 IEEE 24th Intl. Conf. on Data Engineering*. IEEE, 1111–1120.

[28] Nimrod Megiddo and Dharmendra S. Modha. 2004. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer* 37, 4 (2004), 58–65.

[29] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages.

[30] Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. 2016. Parallel Tree Traversal for Nearest Neighbor Query on the GPU. In *45th Intl. Conf. on Parallel Processing*. 113–122.

[31] NVIDIA. 2017. CUDA Programming Guide 9.0. http://docs.nvidia.com/cuda Accessed: 17-05-2019.

[32] NVIDIA. 2017. Volta. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf Accessed: 31-01-2019.

[33] PCI-SIG. 2017. PCI-SIG DevCon 2017 Update. https://pcisig.com/sites/default/files/files/PCI-SIG%20DevCon%202017%20Press%20Deck.pdf Accessed: 23-02-2019.

[34] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 755–766.

[35] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE Intl. Parallel and Distributed Processing Symposium Workshops*. 618–627.

[36] D. Schaa and D. Kaeli. 2009. Exploring the multiple-GPU design space. In *IEEE Intl. Parallel & Distributed Processing Symposium*. 1–12.

[37] Johannes Singler and Benjamin Konsik. 2008. The GNU libstdc++ parallel mode: software engineering considerations. In *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 15–22.

[38] David Taniar and J. Wenny Rahayu. 2002. Parallel Database Sorting. *Inf. Sci.* 146, 1-4 (Oct. 2002), 171–219.

[39] Vassilis J Tsotras and Nickolas Kangelaris. 1995. The snapshot index: an I/O-optimal access method for timeslice queries. *Information Systems* 20, 3 (1995), 237–260.

[40] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.