A Study of Work Distribution and Contention in Database Primitives on Heterogeneous CPU/GPU Architectures

Anonymous Author(s)

ABSTRACT

Graphics Processing Units (GPUs) provide very high on-card memory bandwidth which can be exploited to address data-intensive workloads. To maximize algorithm throughput, it is important to concurrently utilize both the CPU and GPU to carry out database queries. We select data-intensive algorithms that are common in databases and data analytic applications including: (*i*) scan; (*ii*) batched predecessor searches; (*iii*) multiway merging; and, (*iv*) partitioning. For each algorithm, we examine the performance of parallel CPU/GPU-only, and hybrid CPU/GPU approaches.

There are several challenges to combining the CPU and GPU for query processing, including distributing work between architectures. We demonstrate that despite being able to accurately split the work between the CPU and GPU, contention for memory bandwidth is a major limiting factor for hybrid CPU/GPU data-intensive algorithms. We employ performance models that allow us to explore several research questions. We find that while hybrid data-intensive algorithms may be limited by contention, these algorithms are more robust to workload characteristics; therefore, they are preferable to CPU/GPU-only approaches. We also find that hybrid algorithms achieve good performance when there is low memory contention between the CPU and GPU, such that the GPU can perform its operations without significantly reducing CPU throughput.

KEYWORDS

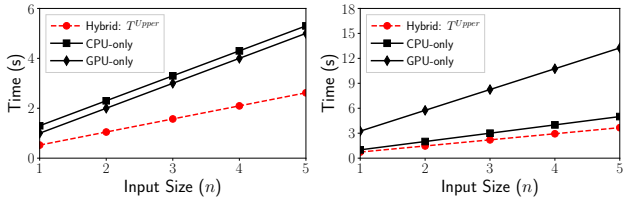
GPGPU, Heterogeneous Systems, Hybrid Algorithms, In-memory Database, Memory-Bound Algorithms, Multiway Merge, Partitioning, Predecessor Search, Scan

ACM Reference Format:

Anonymous Author(s). 2021. A Study of Work Distribution and Contention in Database Primitives on Heterogeneous CPU/GPU Architectures. In *Proceedings of ACM SAC Conference (SAC'21).* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Graphics processing units (GPUs) have been exploited to improve data-intensive algorithm throughput. Algorithms can now be designed to use the CPU or GPU [12, 26], in addition to hybrid CPU/GPU approaches that exploit both architectures [21]. In a shared-memory system with limited resources, the distribution of work and tasks between CPU and GPU architectures has to be carefully considered to optimize resource utilization and algorithmic efficiency.



(a) Even Split (b) Uneven Split

Figure 1: Synthetic example with an (a) near-even split and (b) uneven split of work between the CPU and GPU. T^{Upper} is the upper bound on performance derived by the combined throughput of the CPU-only and GPU-only algorithms.

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 [23] and Nvidia Volta has 900 GiB/s global memory bandwidth [22]). Thus, for data-intensive algorithms, the CPU-GPU interconnect is a performance bottleneck [26]; however, there is an opportunity to exploit the GPU's high on-card memory bandwidth.

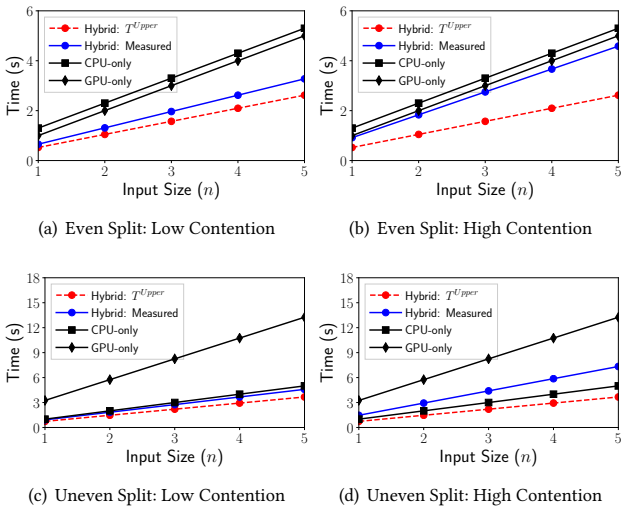
A recent study by Gowanlock et al. [12] proposed a work splitting strategy to assign work to the CPU and GPU for hybrid memory-bound database primitives. However, this paper made several simplifying assumptions about work distribution between the CPU and GPU which we believe are unable to adequately capture the performance of the examined hybrid algorithms. As we will demonstrate with an example later in the introduction, contention for memory bandwidth can be detrimental to hybrid CPU/GPU algorithm performance, thus *we fundamentally disagree with the approach taken by Gowanlock et al.* [12].

While we disagree with the modeling approach used to split the work between the CPU and GPU in Gowanlock et al. [12], the authors proposed a comprehensive testbed of database primitives. In particular, the authors proposed hybrid batched predecessor searches, multiway merging, and k-way partitioning. We summarize each of these algorithms in Section 4. The algorithms use a significant fraction of available main memory bandwidth in the system. Therefore, these primitives provide a good testbed of common database workloads that can be used for examining the impacts of contention on hybrid CPU/GPU algorithm performance.

To elaborate on the prior work and pitfalls of the model proposed by Gowanlock et al. [12], as an illustrative example, we show that there are two major limitations that may hinder hybrid CPU/GPU algorithms that we describe as follows.

SAC'21, March 22-March 26, 2021, Gwangju, South Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/xx.xxx/xxx_x



(a) Even Split: Low Contention (b) Even Split: High Contention

(c) Uneven Split: Low Contention (d) Uneven Split: High Contention

Figure 2: Synthetic example with even and uneven even splits of work between the CPU and GPU for low and high contention scenarios.

First limitation: Splitting the Work. This limitation was considered by Gowanlock et al. [12]. The upper bound on the speedup of a hybrid algorithm over both CPU-only and GPU-only approaches is when the work is evenly split between the two architectures (i.e., each architecture computes half of the total work). Figure 1(a) shows a synthetic example of the response time vs. input size, n, where the CPU-only and GPU-only algorithms achieve nearly the same performance, and where T^{Upper} is the modeled upper bound throughput. The upper bound throughput is simply the combined throughput of the CPU-only and GPU-only algorithms. In this case, there is significant potential for a hybrid algorithm to outperform its CPU-only and GPU-only counterparts. In contrast, Figure 1(b) shows the case where the GPU-only algorithm performs much worse than the CPU-only algorithm. In this case, the upper bound on performance is similar to the CPU-only performance, indicating that at most, a hybrid algorithm will only achieve negligible performance gains[1].

Second limitation: Contention for Resources. This limitation was not considered by Gowanlock et al. [12]. In the case where an even distribution of work between architectures is possible, memory bandwidth contention can significantly reduce performance. This is a major concern for data-intensive algorithms.

Figure 2(a) shows the best case for a hybrid CPU/GPU algorithm where there is an even split and low contention for main memory bandwidth. Here, the measured hybrid algorithm response time is close to the upper bound on performance (T^{Upper}). Figure 2(b) shows that despite a near-even split in work between the CPU and GPU, contention for main memory bandwidth limits the performance of the hybrid algorithm over the GPU-only algorithm.

[1]Note that if the CPU-only algorithm performs much worse than the GPU-only algorithm then this would yield the same upper bound performance and have the same implications.

Figure 2(c) shows that when there is an uneven split, the performance gain of the hybrid algorithm over the CPU-only algorithm is minimal despite low memory contention. And lastly, in Figure 2(d), we find that the hybrid algorithm has worse performance than the CPU-only algorithm due to both the uneven work split and high contention. Consequently, the hybrid algorithm *should not be employed* in this case.

Due to the above limitations, hybrid CPU/GPU approaches do not perform well across all workloads. However, we can selectively employ hybrid algorithms depending on workload characteristics by using performance models that quantify: (i) the upper bound on performance; (ii) contention; and (iii) memory bandwidth saturation.

Major Benefit of Hybrid Algorithms: Depending on workload characteristics, hybrid algorithms only achieve performance gains in certain cases. However, they are *more robust* as they can improve performance in scenarios where the CPU/GPU-only algorithms do not perform well.

Research Questions Answered by the Models: Models are used to answer the following questions: (i) Are common data-intensive workloads amenable to heterogeneous architectures? (ii) Is main memory bandwidth saturated by canonical data-intensive algorithms? (iii) New architectures are heterogeneous and require different algorithms to achieve peak performance. What algorithm properties indicate that they will yield good performance when executed in a hybrid fashion? (iv) To what extent does memory contention degrade the performance of a hybrid algorithm?

As a demonstration of the potential improvement over CPU/GPU-only primitives, we efficiently solve the following problems: (i) scan; (ii) batched predecessor searches; (iii) multiway merging; and, (iv) k-way partitioning. These four data-intensive algorithms are used in several canonical database applications [13, 15, 28, 29, 32]. Following Gowanlock et al. [12], to reduce the memory pressure of the CPU/GPU-only and hybrid approaches, we employ algorithms that are optimal in the well-known external memory (EM) model [1]. This paper makes the following major contributions:

- We employ database primitives that are optimal in the EM model to minimize memory accesses and contention. We show that contention for resources between the CPU and GPU can degrade the performance of hybrid algorithms; thus, I/O efficient algorithms are of paramount importance.
- We propose three models: (i) the upper bound on hybrid algorithm performance; (ii) a model that includes contention to be compared with the upper bound model; and (iii) a model for the CPU-only algorithm that assumes memory bandwidth saturation.
- We show that our upper bound model is very accurate at splitting the work between CPU and GPU architectures. This is important, as an inaccurate split can degrade performance due to load imbalance between architectures.

We summarize the major differences between this paper, and the prior work proposed by Gowanlock et al. [12] as follows:

- Prior work developed a model to split the work between the CPU and GPU. However, that model completely ignored the cost of computation and assumed that all algorithms saturate main memory bandwidth. In this paper, we consider the cost of computation and show that this cost is not always negligible as

previously assumed. Additionally, in contrast to prior work, we show that main memory bandwidth is not saturated in all of the hybrid algorithms.
- Prior work was unable to constrain the upper bound on performance of the hybrid approaches. This paper places an upper bound on performance based on the throughput of the CPU/GPU-only algorithms.
- Prior work assumed a contention-free scenario, where the use of the CPU and GPU did not compete for memory bandwidth. In contrast, this paper focuses on contention and clearly demonstrates that it can degrade the performance of hybrid CPU/GPU algorithms. Thus, prior work made unsatisfactory model assumptions.
- Prior work lacked experimental evidence to support model assumptions. This paper supports assumptions using a set of hypothesis testing experiments.
- Prior work assumed that all algorithms are suitable for a hybrid CPU/GPU execution. Using our models, we are able to detect whether an algorithm should be executed using the CPU and GPU, or whether it should simply use one of the CPU/GPU-only approaches.

This paper addresses several of the major limitations of the pioneering work of Gowanlock et al. [12]. While the efforts of the prior work showed the potential of hybrid CPU/GPU database primitives, this paper provides a thorough treatment of contention, which has several implications for the design of in-memory database systems that exploit emerging architectures.

The paper is organized as follows. Section 2 outlines related work. Section 3 describes the performance models. Section 4 describes the hybrid algorithms we propose. Section 5 demonstrates the effectiveness of the hybrid algorithms and the utility of the models. Section 6 concludes the paper.

2 BACKGROUND & RELATED WORK

In this section, we describe the problem statement and constraints considered in this work, and discuss this paper in the broader context of the related literature.

Problem Statement & Constraints: We consider CPU/GPU-only and hybrid CPU/GPU algorithms. The total response time of an algorithm includes all data transfers to and from the GPU and related overheads. The final result set is stored in main memory. The input and output sizes can exceed the GPU's global memory capacity, which is enabled by partitioning the input data and executing several independent batches. Most GPU algorithms proposed in the literature only include GPU computation time and do not account for data transfers in their evaluations [2, 5, 14, 17, 25]. We account for all data transfers. In many cases, the cost of computation is minor relative to the cost of PCIe data transfers and other host-to-host memory operations that support GPU computation.

GPU-Accelerated Databases & Modeling: Several studies model the execution of workloads on GPUs [3, 18, 19, 24]. Schaa and Kaeli [24] model application latency considering network bandwidth, disk access throughput, and multi-GPU PCIe contention, and achieve good accuracy across six applications. Kothapalli et al. [19] propose a general model of GPU computation that employs other well-known models (BSP [30], PRAM [7], and QRQW [9, 10]). Boyer et al. [3] focus on modeling data transfers, rather than kernel execution time, which is particularly important for data-intensive workloads. Van Werkhoven et al. [31] focus on modeling data transfers between the CPU and GPU where communication can be overlapped in CUDA streams to hide data transfer overhead.

We employ a model used by Shanbhag et al. [26] that assumes main memory bandwidth saturation (this same model assumption was also proposed by Gowanlock et al. [12]).

In contrast to previous work that examines largely GPU-only approaches or compares GPU-only to CPU-only approaches, this paper is focused on understanding splitting the work between CPU and GPU architectures. Therefore, previous work does not consider models that distribute the work between architectures.

Karnagel et al. [16] explore the limitations of splitting workloads between the CPU and GPU, and discuss the effects of resource undersaturation, synchronization overhead, and merging intermediate results into the final result set. They propose a model for splitting the workload, and caution the reader that hybrid CPU/GPU approaches need to be carefully designed if they are to achieve performance gains over CPU-only approaches. Similarly to our approach, they partition the data to be executed on the CPU and GPU and assume that the data resides in main memory at the end of the computation. In contrast to Karnagel et al. [16], this paper examines the impact of memory bandwidth contention and models several scenarios not considered in their work.

Data Transfer Optimizations: Several studies have optimized data transfers between the host and GPU [8, 11, 20]. We employ several data transfer optimizations in Gowanlock and Karsin [11], which include reusing small pinned memory buffers to transfer data between the host and GPU using several CUDA streams.

3 GENERAL OUTLINE FOR MODELING ALGORITHMS

We outline models that apply to all algorithms. Let T_1 and T_2 denote the CPU-only and GPU-only execution time for a given algorithm in seconds (with input size n), respectively. The CPU-only and GPU-only throughput is denoted as $r_1 = n/T_1$ and $r_2 = n/T_2$, respectively; we assume the throughput is independent of input size, n.

We assume that the upper bound throughput of a hybrid CPU/GPU algorithm that splits the work between architectures is the total throughput given by the CPU-only and GPU-only algorithms, denoted as $r_{tot} = r_1 + r_2$. In practice, the upper bound throughput is not achievable, unless there is no contention for resources in the system.

3.1 The Fraction of Work Assigned to the CPU and GPU

The total throughput of the CPU-only and GPU-only algorithms is given by r_{tot}. We compute the fraction of work assigned to the CPU (f) and GPU ($1 - f$) such that we evenly split the work between the CPU and GPU, as follows:

$$f = r_1/r_{tot}. \quad (1)$$

This assumes that the throughput ratio of the CPU to the GPU components of the hybrid algorithm are identical to the ratio of the

CPU-only and GPU-only algorithms (r_1/r_2). Since the hybrid algorithm components are the same as the CPU/GPU-only algorithms, but they work on different data partitions, these throughput ratios are expected to be equal.

3.2 Hybrid Model: Upper Bound

Let T^{Upper} be the modeled upper bound on performance of a hybrid algorithm (with a throughput of r_{tot}), given in seconds as follows:

$$T^{Upper} = n \cdot \ f \cdot r_1^{-1} = n \cdot \ (1-f) \cdot r_2^{-1}, \tag{2}$$

where f and $1-f$ is the fraction of work assigned to the CPU and GPU, respectively. Since both the CPU and GPU are executing concurrently, the model assumes the upper bound throughput $r_{tot} = r_1 + r_2$ is achieved.

3.3 Hybrid Model: Including Contention

Since the upper bound throughput, r_{tot}, is unachievable in practice due to contention for resources in the system, we quantify the difference between the expected time given by r_{tot} and the measured hybrid algorithm time using f, denoted as T_3. The contention factor, c, is given as follows:

$$c = T_3/T^{Upper}. \tag{3}$$

Since T^{Upper} is the modeled upper bound on performance, $T_3 > T^{Upper}$; therefore, $c > 1$. We modify T^{Upper} to include the contention factor as follows:

$$T^{Hybrid} = c \cdot T^{Upper}. \tag{4}$$

3.4 Should the Hybrid Algorithm be Used?

Observe in Equation 4 that depending on the values of f, r_1 (or r_2), and c, the model may yield a response time greater than the CPU/GPU-only algorithms. Since deriving the contention factor (Equation 3) requires measuring T_3, it can be compared to determine whether $T_3 > min(T_1, T_2)$. If this is the case, then the hybrid algorithm should not be used as it leads to performance degradation relative to the CPU/GPU-only algorithms.

3.5 CPU-only Model: Saturated Memory Bandwidth

Gowanlock et al. [12] and Shanbhag et al. [26] model CPU database algorithms assuming they saturate main memory bandwidth. We use the external memory model to ensure that our algorithms minimize loads and stores (Section 1).

This model gives an indication of how memory-bound an algorithm is. If the model accurately captures the performance of the CPU-only algorithm, then this indicates that the algorithm is largely memory-bound and computation is negligible. Otherwise, a non-negligible fraction of the time is spent doing computation. The generalized model is:

$$T^{CPU-only} = \frac{l \cdot n \cdot 8}{\sigma} + \frac{m \cdot n \cdot 8}{\omega}, \tag{5}$$

where σ and ω are the read and write memory bandwidth in bytes/s. In all algorithms, l and m are coefficients of n that describe the number of 8-byte data elements that are read and written. Thus, the modeled time is the total size read (written) divided be the read (write) memory bandwidth. On our platform σ = 43.93 GiB/s and ω = 19.14 GiB/s.

We do not include a similar GPU-only model that examines whether the GPU saturates on-card global memory bandwidth. Because we require the final result set to be stored in main memory, such a model would not capture memory bandwidth saturation as the algorithms are limited by the PCIe v.3 interconnect. Consequently, regarding the primitives that we examine, we assume that computation is (nearly) free on the GPU. In the evaluation, we quantify the time spent performing GPU work.

4 HYBRID ALGORITHMS

There does not exist a standard set of benchmarks for hybrid database primitives. Therefore, we use the algorithms proposed by Gowanlock et al. [12] that introduced hybrid batched predecessor search, multiway merge, and k-way partitioning algorithms. In addition, we have implemented scan as it provides an example of an algorithm that is unlikely to perform well on the GPU when accounting for data transfers. We describe several assumptions below.

Mapping Threads to Tasks: We employ multiple CPU threads for reading/writing data, such that we can saturate memory bandwidth if a given algorithm is memory-bound. We use n_s CUDA streams to saturate PCIe bandwidth and overlap data transfers, where CPU threads orchestrate memory transfers between the host and GPU. We use the data transfer methods of Gowanlock and Karsin [11] that examined the impact of data transfers on hybrid CPU/GPU sorting.

I/O Optimality: All of the algorithms (CPU/GPU-only and hybrid) are optimal in the external memory model. Consequently, the minimum amount of data is transferred between main memory and the CPU/GPU. Since database operations are data-intensive, using the EM model allows for I/O-efficient hybrid CPU/GPU algorithms.

Using Batches: We parallelize the algorithms by partitioning the input into several batches to be executed on the CPU or GPU. We denote the number of batches as n_b, and set n_b to avoid the negative effects of load imbalance. For scan, batched predecessor search, and partitioning we set n_b = 400. For the multiway merge primitive, n_b is a function of k to mitigate overheads (batches may be so small that overheads are non-negligible). Batches enable parallel computation and fit within GPU global memory capacity. Since we examine large input sizes (up to the maximum main memory capacity of the platform), the overhead required to partition the data into batches is negligible.

4.1 Algorithm Test Suite

We rely on algorithms from the literature. For convenience, we briefly describe the hybrid algorithms presented in Gowanlock et al. [12], but refer the reader to that work for more information. Additionally, we illustrate the scan algorithm that was not employed in that work.

4.1.1 Scan. Scan is an operation used extensively by database systems, such as finding the minimum or maximum value in a table. We define scan as follows, and use the *max* function, which requires reading all n elements in a list to find the maximum value

(alternatively, we could use a different function, such as *min*, but the complexity is the same). Let A be an unordered array of n elements. We find $x \in A$ such that for all $y \in A$, $x \geq y$, i.e. $x = max(A)$. We selected scan as it has a high memory access to compute ratio, which makes it a poor candidate for acceleration on the GPU; therefore, it may indicate the limits of hybrid CPU/GPU computation.

CPU-Only Algorithm: The algorithm iterates over array A to find $max(A)$. This loop is trivially parallelized using the maximum reduction in OpenMP, where n elements are read from main memory to the CPU, and $O(1)$ elements are transferred to main memory.

GPU-only Algorithm: We split A into n_b batches of equal size and transfer each batch from the host to the GPU (*HtoD*) using n_s CUDA streams. Then, the maximum of each batch B_i, $i = 1, 2, \ldots, n_b$ is found. Each CUDA stream is assigned a local maximum on the device in global memory denoted as M_j^{local}, and temporary storage for the current maximum being computed M_j^{temp}, where $j = 1, 2, \ldots, n_s$.

Each CUDA stream computes $\frac{n_b}{n_s}$ batches[2]. At each kernel invocation that executes a batch in a stream, the maximum value in B_i is stored in M_j^{temp}. Then, M_j^{temp} is compared to the local maximum found in the stream thus far, and is updated accordingly, i.e., $M_j^{local} = max(M_j^{temp}, M_j^{local})$. Once this has been done for all batches across all streams, the array of local maximums, M_j^{local}, are transferred to the host where the global maximum is computed. Therefore, $max(A) = max(M_1^{local}, M_2^{local}, \ldots, M_{n_s}^{local})$. Thus, n elements are transferred to the GPU, and since $n_s \approx 1$, $O(1)$ elements are transferred back to the host.

Hybrid Algorithm: To combine the CPU and GPU algorithms, we utilize the batching scheme to split A into several batches, which allows us to split the total work between architectures. Each batch contains $\frac{n}{n_b}$ elements. Since $max(A) = max(max(B_1), max(B_2), \ldots, max(B_{n_b}))$, queries can be executed independently on both architectures.

4.1.2 Batched Predecessor Search (BPS). Let A be keys and B be queries both of which are sorted in non-decreasing order. Each key is denoted as a_i, where $i = 1, 2, \ldots, n$, and each query is denoted as b_j, where $j = 1, 2, \ldots, n$. The batched predecessor search (BPS) finds the largest value of i for each $b_j \in B$, such that $a_i \leq b_j$. In our evaluation, we assume $|A| = |B| = n$.

CPU-Only Algorithm: For each $b_j \in B$, the algorithm executes a merge find which finds the index in A without merging [6]. A and B are partitioned into n_b batches, where each processor computes batches of size n/n_b to find the predecessor of each query. The algorithm reads and writes a total of $2n$ and n elements in main memory, respectively.

GPU-only Algorithm: An upper bound binary search is executed on each $b_j \in B$. Unlike the CPU-only algorithm, the GPU algorithm must be able to independently compute the queries, which is not possible with the CPU-only approach using the merge find. The BPS algorithm reads and writes a total of $2n$ and n elements in main memory, respectively.

Hybrid Algorithm: Similarly to scan, we split the work between each architecture using batches. In the case of BPS, A and B are partitioned into n_b value-disjoint batches that can be computed on either architecture. We denote each batch as B_i, where $i = 1, 2, \ldots, n_b$. Based on a given value $a \in A$, we find the pivots in B that split the data. Each batch contains roughly $\frac{n}{n_b}$ elements in A and B.

4.1.3 Multiway Merging (MWM). Takes as input a list, A, of k sublists sorted in non-decreasing order, denoted as S_j, where $j = 1, 2, \ldots, k$, where each sublist is of size $\frac{n}{k}$[3]. The output contains n sorted elements. Furthermore, we assume that k is small such that elements loaded from each sublist do not negatively impact CPU cache utilization (e.g., for very large k, cache utilization may degrade).

CPU-Only Algorithm: We use the MWM provided by the GNU parallel mode extensions [27]. A total of n elements are read and written to/from main memory ($2n$ total).

GPU-only Algorithm: A is divided into n_b batches that contain elements from all k lists. Pivots divide A into value-disjoint batches. Each batch is transferred to the GPU to generate a sorted list, which is transferred back to main memory. Merging on each batch is performed by performing a pairwise merge $k - 1$ times. The output is the concatenation of the output of each batch.

Hybrid Algorithm: As with the previous algorithms, we split the work between CPU and GPU, by assigning a fraction of the n_b batches to each architecture.

4.1.4 Partitioning. Let A be an unsorted list of n elements, that is partitioned into k nearly equal sized value-disjoint buckets ($A_1, A_2, \ldots, A_k$). The lower bounds for partitioning n elements into k buckets is $O(n \log k)$ in the RAM and $O(\frac{n}{B} \log_{M/B} k)$ EM models. Repeatedly partitioning n into $\frac{M}{B}$ buckets (which can be done in a single I/O-efficient scan) achieves the external memory bound.

To enable I/O efficiency, we assign a local cache to each bucket while reading the data. The caches are written to main memory when they reach capacity. Each bucket requires a cache of a size B such that it is partitioned into $\frac{M}{B}$ buckets during a single scan. Multiple scans are required if $k > \frac{M}{B}$. μ is the number of buckets partitioned at each scan.

CPU-only Algorithm: The CPU algorithm performs a series of passes; at each pass, each partition is further split into μ subpartitions. $\lceil \log_\mu k \rceil$ passes are needed. Since the ideal choice of μ depends on the size of the CPU cache system, we empirically determine the choice of μ in Section 5.5. During each pass, each CPU thread computes a subset of the input, A, and stores a thread-local cache for each bucket. As A is scanned, threads write their buckets to shared output in main memory. Each thread maintains μ small caches at each scan (we select 1024 elements per cache). Each of the p threads reads $\frac{n}{p}$ elements and writes $\frac{n}{p}$ elements back at each pass, for a total of $2n$ elements across all threads.

GPU-only Algorithm: k-way partitioning is simplified on the GPU by sorting batches rather than bucketing, as there are efficient sorting libraries for the GPU. First, k pivots are transferred *HtoD*. Next, A is partitioned into n_b batches that are sorted. Then, the GPU determines which portions of the batch belong to each bucket

[2]In each algorithm, for illustrative purposes, we assume without the loss of generality that n_s evenly divides n_b.

[3]For illustrative purposes, and without the loss of generality, we assume k evenly divides n.

using a binary search. When the sorted batch is copied *DtoH*, the data is copied into the associated bucket in main memory.

Hybrid Algorithm: As with the other algorithms, the CPU and GPU are assigned independent batches to compute. The GPU component writes the final buckets after each *DtoH* transfer of a batch has completed. This eliminates overhead from merging two sets of buckets (one for the CPU and GPU) at the end of the computation.

5 EVALUATION

5.1 Experimental Methodology

Our platform contains 2× Intel Xeon E5-2620 v4 CPUs, with 16 total physical cores, at a clock rate of 2.1 GHz, and 128 GiB of main memory, equipped with a GP100 with 16 GiB of global memory. Host code is compiled with the O3 optimization flag using the GNU compiler and parallelized using OpenMP [4]. GPU code is written in CUDA 9.

The selected primitives span a large range of values of f (0.34 – 0.71). The throughput of all algorithms is independent of n, and MWM and k-way partitioning algorithms are dependent on k. These varied properties provide a good testbed for examining a range of data-intensive workloads.

We use 8-byte data elements and all results are averaged over 5 trials. All preprocessing work, such as generating batches is included in the response time. For the batched predecessor search, scan, and partitioning algorithms, n_b=400 is selected. MWM uses a batch size as a function of k; otherwise, the batches may be too small which would add unnecessary overhead.

We compare the performance of CPU-only, GPU-only, and hybrid algorithms for each database primitive. We outline their configurations as follows.

- **CPU-only:** 16 threads are executed corresponding to the number of physical cores on our platform. If an algorithm is memory-bound this allows it to saturate main memory bandwidth, and if an algorithm is compute-bound, it allows it to utilize all of the CPU cores.
- **GPU-only:** n_s=8 streams with 8 CPU threads are used to enable saturating bi-directional memory bandwidth over PCIe. Each stream uses pinned memory as a staging buffer of size 8 MiB to copy the data *HtoD* or *DtoH*. The small size of the buffer reduces allocation costs [11].
- **Hybrid:** The combined CPU/GPU-only algorithms above with 24 total CPU threads. We oversubscribe the system with more threads than physical cores to permit memory bandwidth saturation and to exploit all CPU cores.

5.2 Scan

We use scan as an example of a memory-bound algorithm that when executed using the CPU and GPU may not offer any performance advantage over the CPU-only algorithm. Using the recipe in Section 3, we obtain T_1=3.313 and T_2=8.263 for n=4×10^9. From Equation 1, we obtain f=0.71. Executing the hybrid algorithm with f=0.71, we obtain T_3=3.449 and c=1.47. Therefore, since $T_3 > min(T_1, T_2)$, the hybrid algorithm offers no performance gain over the CPU-only algorithm. Consequently, we do not show response times and models for scan, since the hybrid algorithm does not offer a performance advantage. Scan is similar to that shown in Figure 1(b), where either the CPU or GPU performs most of the computation, which limits the potential of a hybrid algorithm.

5.3 Batched Predecessor Search

Using the recipe in Section 3, we model BPS using T^{Upper}, T^{Hybrid}, and $T^{CPU-only}$. We measure the CPU-only and GPU-only algorithms for n=3×10^9 (the median value of n examined). The CPU-only and GPU-only execution time is T_1=7.122 s, and T_2=7.804 s, respectively. Using Equation 1, and computing r_1 and r_{tot}, we obtain f=0.52. Using Equation 3, and executing the hybrid algorithm with f=0.52, we obtain T_3=4.750, yielding c=1.28. The values of f and c are used in Equations 2 and 4 to compute T^{Upper} and T^{Hybrid} for varying n. Since $T_3 < min(T_1, T_2)$ we expect that the hybrid algorithm outperforms the CPU/GPU-only algorithms.

The CPU-only algorithm reads $2n$ elements and writes n elements from main memory, and in Equation 5 we set l=2 and m=1; therefore, $T^{CPU-only} = \frac{2 \cdot n \cdot 8}{\sigma} + \frac{n \cdot 8}{\omega}$.

Figure 3 plots the response time vs. n for BPS, illustrating the CPU-only, GPU-only and hybrid algorithm response times, the hybrid modeled upper bound, hybrid model with contention, and modeled CPU-only algorithm assuming saturated memory bandwidth. We observe that there is a non-negligible difference between T^{Hybrid} and T^{Upper}. Since the contention factor c=1.28, we expect that contention would negatively impact performance. Interestingly, comparing the measured CPU-only execution time to the model, $T^{CPU-only}$, we find that the model significantly underestimates the response time. This indicates that the algorithm performs significant computation. Also, it explains why the hybrid algorithm is able to achieve a speedup over the CPU-only algorithm (of up to 1.56×), as memory bandwidth must not be fully saturated to allow the GPU to perform its memory operations and achieve reasonable performance gains. Furthermore, observe that the load imbalance between the CPU and GPU components is fairly low ($\leq$ 25%), indicating that the model yields a good value of f[4].

Note that the load imbalance varies with n in Figure 3 due to non-uniform memory accesses (NUMA) that cause variation in response times. For example, at n=10^9, the response time is 1.71 s with σ = 0.071 across the time trials. However, if we disable NUMA and only use a single CPU socket (8 cores), the response time is 2.59 s with σ = 0.019; thus, the standard deviation decreases at the expense of reduced performance. In all future experiments that report the load imbalance, we omit showing the performance with NUMA disabled and associated standard deviation, as results are similar.

5.4 Multiway Merge

We use the outline in Section 3 to develop the models. Since the complexity of MWM is $O(n \log k)$, we need separate time measurements for each value of k. For the following measurements we use n = 4×10^9 and set $n_b = \frac{3200}{k}$ due to the amount of memory required to perform MWM on the device[5]. For k=2, T_1=4.532, T_2=5.663,

[4]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.

[5]Because we do pairwise merging, for each CUDA stream we allocate memory for $2 \cdot k \cdot \frac{n}{n_b}$ 64 bit integers. Therefore, the value of n_b is a function of k rather than a constant.

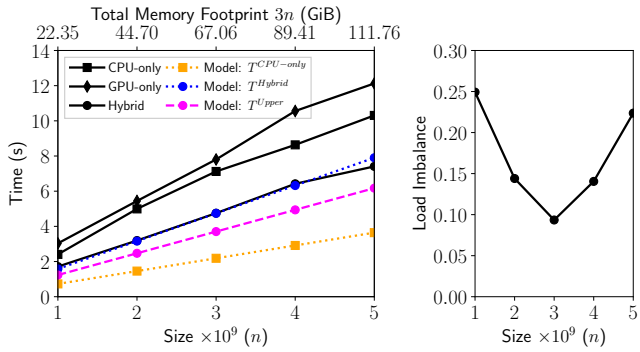


Figure 3: *Left:* Response time vs. input size (n) comparing CPU-only, GPU-only, and hybrid BPS algorithms, T^{Hybrid}, T^{Upper}, and $T^{CPU-only}$ where the total memory footprint, $3n$, is plotted in GiB on the top horizontal axis. *Right:* Load imbalance of the hybrid algorithm on the left.

f=0.56, T_3=3.458, and c=1.36. For k=8, T_1=9.072, T_2=5.949, f=0.40, T_3=4.631, and c=1.28. For k=32, T_1=14.093, T_2=7.304, f=0.34, T_3=6.026, and c=1.26. For all values of k, since $T_3 < min(T_1, T_2)$, we expect that the hybrid algorithm outperforms the CPU/GPU-only algorithms.

The CPU-only algorithm reads and writes n to and from main memory. Therefore, using Equation 5, we set l=m=1, and obtain $T^{CPU-only} = \frac{n \cdot 8}{\sigma} + \frac{n \cdot 8}{\omega}$.

In Figure 4, we show runtimes for our CPU-only, GPU-only, and Hybrid algorithms along with the modeled values $T^{CPU-only}$, T^{Hybrid}, and T^{Upper}. We show $k \in \{2, 8, 32\}$ in Figure 4(a), (b), and (c), respectively.

When k=2 we find the runtimes of the CPU-only and GPU-only algorithm are quite similar. Thus, we expect that an even split of the work between architectures will yield a moderate speedup over either of the single-architecture algorithms. The low load imbalance shown in Figure 4(a) shows that this is the case. Consequently, we find an average speedup of 1.39× over the CPU-only algorithm, and an average speedup of 1.69× over the GPU-only algorithm. Despite this, we observe the non-negligible impact of contention (c = 1.36) means T^{Hybrid} is much larger than T^{Upper}, and peak performance cannot be achieved.

In Figure 4(b), the results for k=8 are shown. We note the degradation in CPU-only performance due to the non-constant factor in the complexity of MWM. $T^{CPU-only}$ illustrates the impact of this factor; our generalized CPU-only model assumes that computation is free, yielding fixed values of $T^{CPU-only}$ across different values of k. The performance of the GPU-only algorithm is independent of k, indicating the bandwidth of the PCIe interconnect is a bottleneck for performance. Observe the low load imbalance across each value of n, demonstrating that our model's value of f=0.40 evenly distributes the batches across different architectures. We observe that the hybrid algorithm achieves an average speedup of 1.82× over the CPU-only algorithm, and an average speedup of 1.28× over the GPU-only algorithm. Similar to when k=2, contention

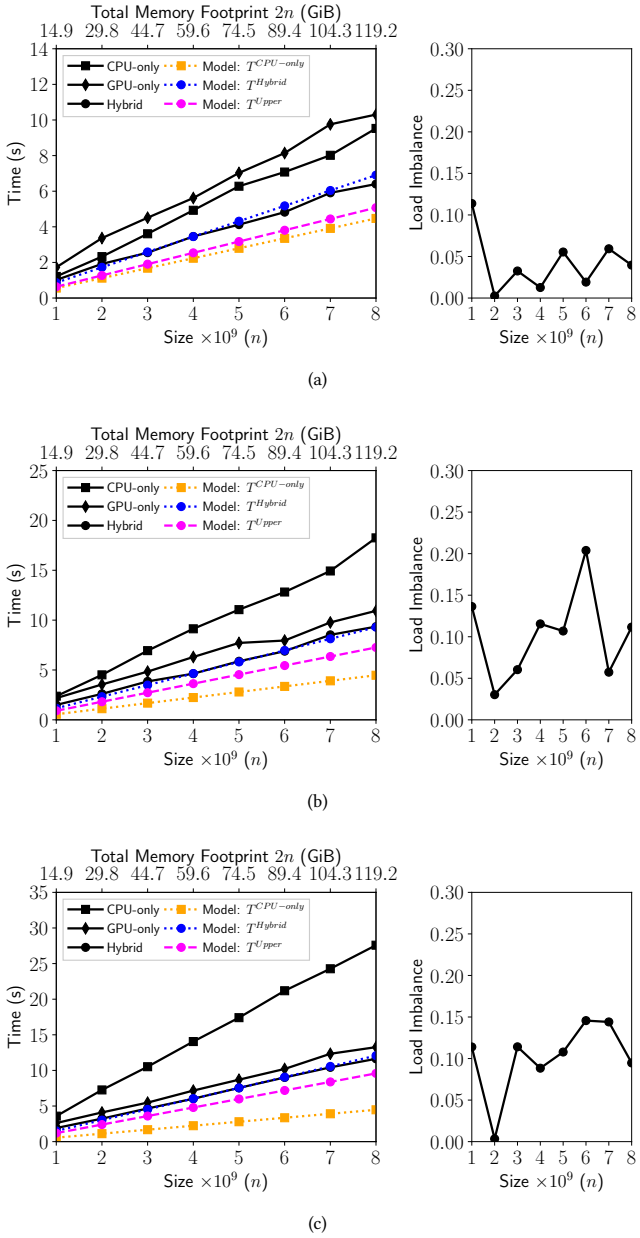


Figure 4: *Left:* Response time vs. input size (n) comparing CPU-only, GPU-only, and hybrid MWM algorithms, T^{Hybrid}, T^{Upper}, and $T^{CPU-only}$ where the total memory footprint, $2n$, is plotted in GiB on the top horizontal axis. *Right:* Load imbalance of the hybrid algorithm on the left. In (a), (b), and (c), we show k=2, 8, and 32, respectively.

is a non-negligible factor in runtime performance; thus, optimal performance, T^{Upper}, is not achieved.

Table 1: Measured times and computed parameters for partition with n=3 × 10^9, and μ=32 for the CPU algorithm.

k	T_1	T_2	f	T_3	c
256	12.07	8.97	0.426	6.86	1.33
1024	12.51	9.11	0.421	6.91	1.32
4096	12.97	9.26	0.419	7.03	1.34

When k=32, the $O(n \log k)$ complexity further degrades CPU-only performance, and our model yields f=0.34. With a large majority of the work being done on one of the architectures, MWM resembles an algorithm shown in Figure 1(b), and a negligible speedup over the GPU-only architecture is to be expected. However, since $T_3 < min(T_1, T_2)$, we expect that the hybrid algorithm outperforms the CPU-only and GPU-only algorithms, as shown in Figure 4(c). Across all values of n, an average speedup of 2.26× over the CPU-only algorithm, and an average speedup of 1.20× over the GPU-only algorithm is shown. The low load imbalance indicates that f=0.34 determines an appropriate batch distribution between the CPU and GPU. As the memory contention factor c=1.26 is large, there is a notable difference between T^{Hybrid} and T^{Upper}.

5.5 Partitioning

As discussed in Section 4.1.4, 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. Since the ideal choice of μ depends on the CPU hardware features, we measure this value empirically on our platform and determine that μ=32 provides the best average performance on a range of k values. Thus, on our platform we use μ=32 for all experiments.

As with MWM, the complexity of partition depends on k (the CPU-only algorithm is $O(n \log_\mu k)$). Thus, we measure execution times and compute parameters for several values of k, using n=3 × 10^9 and μ = 32 for all measurements, with results listed in Table 1. We see that the GPU performs partitioning somewhat faster across all values of k. The ratio of CPU to GPU performance remains somewhat consistent, so f and c are approximately the same for all values of k ($f \approx 0.42$ and $c \approx 1.32$).

For more detailed analysis, we focus on the case where k=1024 and compute $T^{CPU-only}$ using Equation 5. For this case, l=m= $\log_\mu k$=2, therefore $T^{CPU-only} = \frac{2 \cdot n \cdot 8}{\sigma} + \frac{2 \cdot n \cdot 8}{\omega}$. Figure 5 plots the CPU-only, GPU-only, and hybrid response times vs. n for this case. We observe a substantial difference between the modeled CPU-only ($T^{CPU-only}$) and measured response times, indicating that partitioning is not memory bound on the CPU. Additionally, the measured hybrid response time achieves a respectable fraction of the upper bound throughput given by T^{Upper}.

5.6 GPU Computation Time: Effect of Contention and Validation of Model Assumptions

We have made several model assumptions in Section 3 that are used to examine the research questions in Section 1. We describe and validate two assumptions as follows.

Assumption 1: The ratio of the CPU-only to GPU-only throughput is the same as the ratio of the CPU to GPU throughput of the hybrid algorithm. This assumption allows us to select the fraction of work assigned to the CPU and GPU (f), and develop the upper bound hybrid model (T^{Upper}). It is possible that the throughput ratios are not the same, due to unexpected performance behavior that arises with contention for memory bandwidth (e.g., one hybrid component is more resilient to contention than the other).

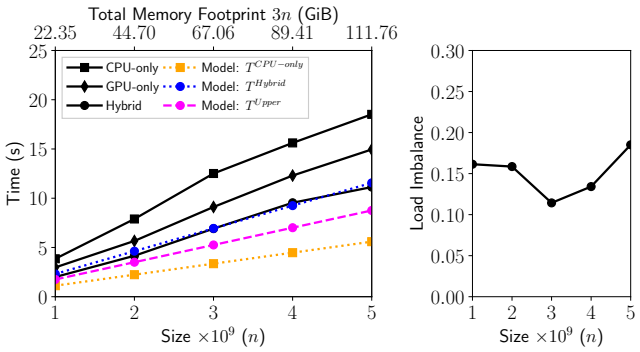


Figure 5: *Left:* Response time vs. input size (n) comparing CPU-only, GPU-only, and hybrid partitioning algorithms, T^{Hybrid}, T^{Upper}, and $T^{CPU-only}$ where the total memory footprint, 3n, is plotted in GiB on the top horizontal axis. *Right:* Load imbalance of the hybrid algorithm on the left.

We hypothesize that contention from the CPU component of the hybrid algorithm should decrease the rate at which GPU kernels can be launched, as each kernel requires memory operations be performed before execution (i.e., copying data into pinned memory buffers, sending the data to the GPU, and copying data back to the host). By comparing the total fraction of time executing kernels between the GPU-only and GPU component of the hybrid algorithm, we can observe whether contention is decreasing the fraction of time spent executing kernels in the hybrid algorithm.

The fraction of the total time executing kernels for the hybrid algorithm is T_k/T_{GPU}, where T_k is the total time executing kernels and T_{GPU} is time total time executing the GPU component of the hybrid algorithm. Regarding the GPU-only algorithm, the fraction is simply T_k/T, where T is the total algorithm response time. Table 2 shows the fraction of time performing computation across all algorithms for median values of n in the experiments. Across all algorithms, the fraction of time performing computation in the hybrid algorithm is less than the GPU-only algorithm. This is because memory contention decreases the rate at which kernels can be launched. Thus, contention reduces the GPU's ability to perform computation.

Observe that the ratio of time performing computation in the GPU-only to hybrid algorithms is roughly consistent with the modeled values of c in all algorithms (i.e., for BPS, the ratio is 1.25 and c=1.28). We expect these to be consistent if the assumption was correct; therefore, we believe that this model assumption is verified through this experiment.

Assumption 2: Computation on the GPU is assumed to be nearly negligible because the cost of transferring data to and from the GPU dwarfs GPU computation cost. This assumption can be validated by simply examining the fraction of time computing on the GPU

Table 2: Fraction of time spent executing kernels comparing GPU-only to hybrid algorithms for each primitive (excluding scan), the ratio of the GPU-only to hybrid computation times, and the modeled values of c that indicate contention.

Algorithm	n ($\times 10^9$)	GPU-only	Hybrid	Ratio	c
BPS	3	0.065	0.052	1.25	1.28
MWM (k=2)	4	0.030	0.024	1.25	1.36
MWM (k=8)	4	0.077	0.065	1.18	1.28
MWM (k=32)	4	0.295	0.235	1.26	1.26
Partition (k=1024)	3	0.096	0.063	1.53	1.32

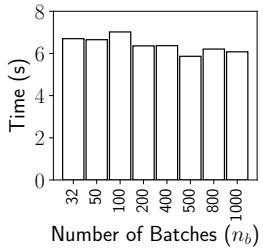


Figure 6: Total BPS response time vs. the number of batches, n_b, for n=4.0 × 10^9. We omit showing the other algorithms as results are similar.

relative to the total execution time of the GPU-only algorithm. Table 2 supports this assumption, as GPU computation is a small fraction of the total response time in each algorithm.

5.7 Performance Impact of the Number of Batches

We split the work into several independent batches. Each batch computes some fraction of the total work, and as explained in Section 4, we set n_b=400 in all experiments except for MWM, where we set n_b=$\frac{3200}{k}$ (Section 5.4).

Figure 6 plots the response time of BPS vs. n_b. The response time roughly decreases with increasing n_b. Thus, a small number of batches degrades performance due to load imbalance. However, the response time is roughly independent of n_b when $n_b \geq 400$ indicating that the use of many independent batches does not degrade performance.

5.8 Accuracy of Splitting the Work

Figure 7 illustrates how well the model splits the work between the CPU and GPU by executing each primitive that should employ the hybrid algorithm (BPS, MWM, and partitioning) for varying values of the fraction of work computed on the CPU (f). We use n=4×10^9 for all algorithms. In the figure, T_{CPU} and T_{GPU} correspond to the times at which the CPU and GPU complete computing their batches. We find that across all algorithms, the value of f shown by the vertical dashed line, yields an efficient distribution of work between architectures, as each architecture completes its work at roughly the same time. Additionally, this experiment validates our model assumption that the ratio of the throughput achieved on CPU-only to GPU-only algorithms is consistent with the ratio of the CPU and GPU components of the hybrid algorithms.

5.9 Discussion: Comparison with Prior Work

As discussed in Section 1, the prior work of Gowanlock et al. [12] only considered a model that splits the work between the CPU and GPU, which (i) assumed that algorithms saturate main memory bandwidth; (ii) assumed that computation was free; and (iii) did not consider contention. We find that all of these model assumptions are unsupported by experimental evidence. In contrast, we find the following: (i) memory bandwidth was not saturated on the three hybrid database primitives explored in their work, (ii) computation has a non-negligible cost; and, (iii) contention plays a significant role in degrading hybrid algorithm performance.

Table 3: Speedup of the hybrid over CPU-only and GPU-only algorithms averaged across all values of n in Figures 3, 4, and 5.

Algorithm	f	c	CPU-only	GPU-only
BPS	0.52	1.28	1.44×	1.68×
MWM (k=2)	0.56	1.36	1.39×	1.69×
MWM (k=8)	0.40	1.28	1.82×	1.28×
MWM (k=32)	0.34	1.26	2.26×	1.20×
Partition (k=1024)	0.42	1.32	1.79×	1.36×

Figure 6 in Gowanlock et al. [12] shows high load imbalance in the multiway merge algorithm due to their assumption that computation is free. In contrast, our proposed model is able to accurately predict the response time of the multiway merge algorithm. From our analysis, we know that computation is minimal when k=2, but has a significant impact when k=32; therefore, this explains the poor load imbalance in their work.

Comparing Figure 7 in our paper to Figure 9 in Gowanlock et al. [12], we find that their model achieves a good distribution of work between architectures. We believe that their model is accurate for the wrong reasons. Overall, comparing the load imbalance of the individual algorithms between our work and Gowanlock et al. [12], we find that our algorithms typically achieve lower load imbalance due to the abovementioned factors.

6 CONCLUSIONS

We discuss the questions outlined in the introduction.

Are data-intensive workloads amenable to heterogeneous architectures? We examined four data-intensive algorithms that are at first glance unsuitable for execution on the GPU. Our results show that there are substantial performance benefits to using a hybrid CPU/GPU approach (see Table 3).

Is main memory bandwidth saturated by canonical algorithms? Based on the CPU-only model that assumed main memory bandwidth saturation, the BPS, MWM, and partitioning primitives did not saturate memory bandwidth. The scan algorithm is able to saturate main memory bandwidth, but it was unsuitable for a hybrid execution.

What algorithm properties indicate that they will yield good performance when executed in a hybrid fashion? From Table 3, MWM with k=32 has the lowest value of f and has the smallest contention factor c, indicating that it is highly amenable to execution on the GPU. In contrast, MWM with k=2 yields the highest value of f in Table 3 and has the largest contention factor c, which indicates that it is more amenable to the CPU. Therefore, f and c are correlated. We find that algorithms with random memory accesses, that perform significant computation, and that have low contention, are able to best exploit the GPU. Algorithms with linear memory

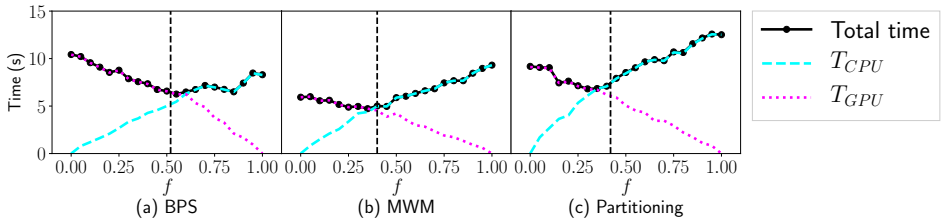


Figure 7: Accuracy of splitting the work shown as the total response time vs. the fraction of work assigned to the CPU, f. A value of f=0 indicates that all work is assigned to the CPU, whereas f=1 indicates that all work is assigned to the GPU. The modeled value of f is shown as the vertical dashed line. T_{CPU} and T_{GPU} correspond to the times at which the CPU and GPU complete computing their batches. MWM is configured with k=8; partitioning is configured with μ=32 and k=1024.

accesses, such as scan perform best on the CPU. Therefore, a hybrid algorithm requires relatively low memory contention, such that the GPU can perform its operations without substantially reducing CPU throughput, and we observe this occurs when $f \lesssim 0.5$. If f is too low, then the contribution of the CPU to the throughput of the hybrid algorithm is minimal.

To what extent does memory contention degrade the performance of a hybrid algorithm? Excluding scan, we found that the average slowdown across all values of n of the hybrid algorithm relative to the upper bound modeled throughput was 0.73×–0.80×. Thus, contention has a non-negligible impact on performance.

Algorithms that are currently bound by the host-device interconnect, such as PCIe, will see major performance improvements with new interconnects, such as NVLink. In hybrid CPU/GPU algorithms, the GPU will be able to compute a larger total fraction of the work. Despite this increase in GPU throughput, these new interconnects will quickly saturate main memory bandwidth. For example, only two GPUs are needed to saturate main memory bandwidth with NVLink 2.0 [20] which will increase memory pressure. This will limit the performance of multi-GPU systems for data-intensive workloads, as GPUs will compete for main memory bandwidth similarly to CPU/GPU contention in this paper. As a result, the bottleneck will shift from the host-device interconnect to main memory bandwidth for many database primitives.

REFERENCES

[1] Alok Aggarwal and Jeffrey Vitter. 1988. The input/output complexity of sorting and related problems. *CACM* 31, 9 (1988), 1116–1127.

[2] 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 *Proc. of the 32nd IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 1070–1079.

[3] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. 2013. Improving GPU performance prediction with data transfer modeling. In *IEEE Intl. Symp. on Parallel & Distributed Processing, Workshops and Phd Forum*. 1097–1106.

[4] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan Kaufmann.

[5] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proc. of the ACM Intl. Conf. on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. 46–57.

[6] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. 2009. Using Graphics Processors for High Performance IR Query Processing. In *Proc. of the 18th Intl. Conf. on World Wide Web*. 421–430.

[7] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proc. of the tenth annual ACM Symp. on Theory of computing*. 114–118.

[8] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proc. of the 2018 Intl. Conf. on Management of Data* (Houston, TX, USA). 1603–1618.

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

[10] 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.

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

[12] Michael Gowanlock, Ben Karsin, Zane Fink, and Jordan Wright. 2019. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *Proc. of the 15th Intl. Workshop on Data Management on New Hardware*. Article 7, 11 pages.

[13] 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.

[14] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. 2005. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, Vol. 24. 547–555.

[15] Yannis Ioannidis. 2003. Approximations in database systems. In *Intl. Conf. on Database Theory*. Springer, 16–30.

[16] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2016. Limitations of intra-operator parallelism using heterogeneous computing resources. In *East European Conf. on Advances in Databases and Information Systems*. Springer, 291–305.

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

[18] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 31–42.

[19] 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 *IEEE Intl. Conf. on High Performance Computing*. 463–472.

[20] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *To appear in Proc. of SIGMOD*.

[21] 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.

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

[23] PCI-SIG. 2017. PCI-SIG DevCon 2017 Update. https://pcisig.com/sites/default/files/files/PCI-SIG%20DevCon%202017%20Press%20Deck.pdf Accessed: July 16, 2020.

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

[25] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. 2007. Scan primitives for GPU computing. In *Graphics hardware*. 97–106.

[26] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics (Extended Version). *arXiv preprint arXiv:2003.01178* (2020).

[27] Johannes Singler and Benjamin Konsik. 2008. The GNU libstdc++ parallel mode: software engineering considerations. In *Proc. of the 1st Intl. workshop on Multicore software engineering*. 15–22.

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

[29] 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.

[30] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[31] Ben Van Werkhoven, Jason Maassen, Frank J Seinstra, and Henri E Bal. 2014. Performance models for CPU-GPU data transfers. In *IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing*. 11–20.
[32] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*. IEEE, 224–234.