# Applicability of GPU Computing for Efficient Merge in In-Memory Databases

Jens Krueger, Martin Grund, Ingo Jaeckel, Dr. Alexander Zeier, Hasso Plattner
Hasso Plattner Institute for IT Systems Engineering
University of Potsdam
Potsdam, Germany
{jens.krueger, martin.grund, ingo.jaeckel, alexander.zeier, hasso.plattner}@hpi.uni-potsdam.de

## ABSTRACT

Column oriented in-memory databases typically use dictionary compression to reduce the overall storage space and allow fast lookup and comparison. However, there is a high performance cost for updates since the dictionary, used for compression, has to be recreated each time records are created, updated or deleted. This has to be taken into account for TPC-C like workloads with around 45% of all queries being transactional modifications. A technique called differential updates can be used to allow faster modifications. In addition to the main storage, the database then maintains a delta storage to accommodate modifying queries. During the merge process, the modifications of the delta are merged with the main storage in parallel to the normal operation of the database. Current hardware and software trends suggest that this problem can be tackled by massively parallelizing the merge process.

One approach to massive parallelism are GPUs that offer order of magnitudes more cores than modern CPUs. Therefore, we analyze the feasibility of a parallel GPU merge implementation and its potential speedup. We found that the maximum potential merge speedup is limited since only two of its four stages are likely to benefit from parallelization. We present a parallel dictionary slice merge algorithm as well as an alternative parallel merge algorithm for GPUs that achieves up to 40% more throughput than its CPU implementation. In addition, we propose a parallel duplicate removal algorithm that achieves up to 27 times the throughput of the CPU implementation.

## 1. INTRODUCTION

In modern enterprise applications the available structured data builds a genuine source for data analysis. However, due to the high update rate it becomes difficult for analytical systems to keep up with the continuous insert stream. As a result such systems use explicit buffering strategies to batch results and apply them together. An optimal system supports this continuous update stream with instant visibility of the operations and provide the required analytical performance on up-to-date data.

As a basis for this approach an in-memory column store database with a differential buffer can be used, e.g. as done in MonetDB X100 [2], C-store [30], and HYRISE [9]. However, the process of applying the differential buffer to the main data partition needs to be optimized, since the overall performance is limited by two factors: The update rate into the write-optimized differential buffer and the performance of the process that merges the buffer with the main partition. Only when the overall update rate is high enough to sustain the continuous update stream the system is operable and remains online.

One of the most interesting optimizations to this process is the parallelization of the merge process. While current server systems are equipped with up to 8-way 12 core CPUs achieving a total number of 96 cores, modern graphic processing units have hundreds of cores that can be used simultaneously. For example, only one of the NVIDIA Tesla C2050 boards we used for our evaluation offers up to 448 CUDA cores that can be programmed. Multiple GPU boards can be attached to a single system to provide an even higher degree of parallelism. In this paper we present an optimized algorithm, implemented in our prototype system [9] for merging the differential buffer with the main partition, which is explicitly tailored to GPUs and evaluates the feasibility of this approach.

We propose a parallel dictionary slice merge algorithm as well as an alternative parallel merge algorithm for GPUs that achieves up to 40% more throughput than its CPU implementation. In addition, we propose a parallel duplicate removal algorithm that achieves up to 27 times the throughput of the CPU implementation.

The remainder of the paper is structured as the following. Section 2 explains the general merge algorithm, Section 3 presents the modifications of the algorithm for GPUs, followed by an evaluation in Section 4. The paper concludes with a short discussion on future work in Section 5 and related work presented in Section 6 and a conclusion (Section 7).

## 2. MERGE

To support analytical reporting capabilities, traditional ERP systems have to utilize pre-processed data and apply caching at various levels. This, however, limits the range of possible reports. In addition, the reported data is likely to be outdated due to the pre-processing. In contrast to the current established transactional enterprise databases, in-memory databases are optimized for read intensive workloads to support agile decision making based on real-time reporting capabilities. Since the majority of tables in typical enterprises contains a limited number of distinct values, the data is compressed and stored in columnar fashion to reduce the query execution time and fit all data into main memory. While compression improves the query processing performance, it rises the cost of updates since they force the compression to be rebuilt.

The recompression of the database is problematic for transactional workloads, where modifying queries can become prohibitively slow during the process. One approach, applied in MonetDB X100 [2], C-store [30], and HYRISE [9], is to batch modifications in a dedicated differential buffer - the delta storage. This buffer is periodically merged with the compressed main storage while the database is running, i.e. the compression of the database is periodically rebuilt. This section describes the process of periodically combining main and delta as the merge algorithm.

On the other hand, parallelism is the future of computing [17, 31, 4]. The power wall forces processor manufacturers to gain performance through the addition of cores rather than increasing clock frequencies. GPUs are highly parallel processors that have proved to be able of outperforming CPUs in a number of applications by up to two orders of magnitude [22, 8, 29, 26, 5]. In this paper, we discuss to which extend the performance of the merge algorithm can be improved through an implementation optimized for the level of parallelism provided by modern GPUs. Furthermore, we propose GPU dictionary merge implementations.

## 2.1 Merge Algorithm Requirements

The merge algorithm has to run while the system is running, i.e. asynchronously. Thus, it must not interfere with the operation of other parts of the database. In general, the merge process should not reduce the overall throughput of the database. If a slowdown of the operation of the database cannot be avoided, the merge should be finished as soon as possible to (1) reduce the time of decreased performance and (2) to free the occupied main memory and computational resources.

During the merge process, the consistency of the data still has to be guaranteed. The merge process must not violate the consistency of the rest of the system's data.

To support consistent reads, lookups have to be done on main *and* the delta. As long as the delta storage is kept small, reads involving the delta storage are expected to be fast. As more updates are stored in the delta, the size of uncompressed data grows and read operations, e.g. selects and joins, get slowed down increasingly. Thus, in analytical read intensive workloads the performance degrades as the delta storage gets bigger.

The size of the delta influences the run-time of the merge process itself as well. Thus, merging main and delta frequently may keep the merge runtime short and reduce the impact to other parts of the system at a minimum. In addition, read operations are performed more likely on compressed data when the amount of uncompressed data is kept small. Consequently, frequent merging may improve the performance for analytical workloads.

The merge process may be triggered (1) implicitly by exceeding a specified threshold, e.g. the memory consumption of the delta, the size of the delta log, the number of tuples, or (2) explicitly by issuing a SQL command. In general, merging should be performed when the performance penalty for read queries, e.g. selects and joins, gets too high.

| | |
|---|---|
| $C^j$ | Column $j$ of table that is currently merged. |
| $M^j$ | Main partition of $C^j$ |
| $D^j$ | Delta partition of $C^j$ |
| $U_M^j$ | Sorted dictionary containing the unique values of $C^j$ of main storage. |
| $U_D^j$ | Sorted dictionary containing the unique values of $C^j$ of the delta storage. |
| $U_{M+D}^j$ | Sorted dictionary containing the unique values of $C^j$ of main and the delta storage, i.e result of step 1. (b). |
| $K^{ij}$ | The $i^{th}$ value of the $j^{th}$ column in uncompressed form. |
| $K_C^{ij}$ | Compressed representation of $K^{ij}$. |
| $b^j$ | Maximum size of bits required to represent a value of the $j^{th}$ column. |
| $r$ | Number of rows in the current table. |
| $d$ | Number of rows in the delta of the current table. Note that this is independent of $j$, i.e. it is the same for all attributes of a table. |
| $u_M^j$ | Number of unique values in the $j^{th}$ column of main, i.e. size of the dictionary $u_M^j = \|U_M^j\|$. |
| $u_D^j$ | Number of unique values in the $j^{th}$ column of the delta, i.e. size of the delta dictionary $u_D^j = \|U_D^j\|$. |
| $u_{M+D}^j$ | Number of unique values in the $j^{th}$ column of the delta, i.e. size of the merged dictionary $u_{M+D}^j = \|U_{M+D}^j\| = \|U_M'^j\|$. |

**Table 1: The notation used to describe the merge algorithm.**

## 2.2 The Merge Algorithm

This section explains the merge algorithm as it is described for HYRISE in [12, 19]. Other column-oriented, in-memory databases apply different approaches, see section 6. We will use the notation shown in table 1 throughout this section. To distinguish from old entities, we add an apostrophe $'$ to all entities that are created as a result of merge. Hence, $M'^j$ denotes the new main storage and $U_M'^j$ describes the new dictionary. Additionally, the notation $|A|$ denotes the cardinality of the set $A$. Thus, $|M^j|$ denotes the number of elements in the $j^{th}$ main column.

During the merge process, a new dictionary $U_M'^j$ is created which is used to compress the old main storage $M^j$ and delta $D^j$ to create a new main $M'^j$. To complete the merge process, this step has to be applied to all columns of a table.

(1) **Merge dictionaries**

(a) **Build dictionary** $U_D^j$ by extracting the unique values of the delta $D^j$. Since HYRISE uses a cache sensitive B+ tree (CSB+ [20]) to represent the data stored in $D^j$, this is reduced to a linear traversal of the leaves of the underlying tree structure. Thus, the runtime is determined by the number of unique values in the delta, i.e. the number of leaves of the CSB+ tree $\mathcal{O}(|U_D^j|)$.

(b) **Merge main and delta dictionaries**, i.e. create a sorted dictionary $U_{M+D}^j$ that contains the unique values of the main dictionary $U_M^j$ and the dictionary of the delta $U_D^j$ created in the previous step. The created dictionary $U_{M+D}^j$ will be the new dictionary $U_M'^j$ for the new main storage $M'^j$, i.e. $U_{M+D}^j = U_M'^j$.

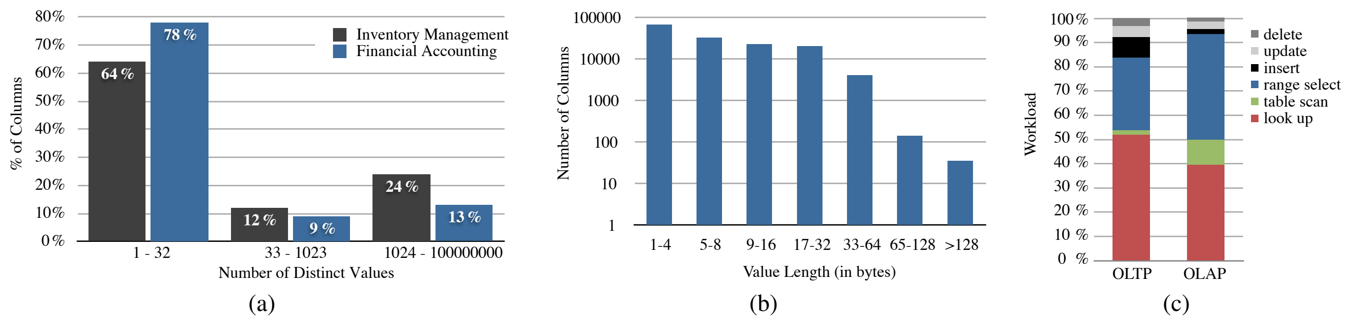


**Figure 1: (a) Fraction of columns with different numbers of unique values. (b) Distribution of value length ranges across columns. (c) Distribution of read and modifying queries in OLAP and OLTP workloads.**

Due to the assumption that both $U_M^j$ and $U_D^j$ are sorted, the runtime of this step is limited by the number of comparisons required to create the new dictionary $U_{M+D}^j$. When both unique value containers, e.g. arrays, are accessed and compared using iterators, at most $|U_M^j| + |U_D^j|$ comparisons are needed to create $U_{M+D}^j$. The overall computational complexity is, therefore, $\mathcal{O}(|U_M^j| + |U_D^j|)$.

(2) **Updating compressed values**

(a) **Compute new compressed value length.** This step calculates the number of bits necessary to represent $|U_{M+D}^j|$ distinct values, i.e. the new compressed value length. The number has to be calculated again, since the cardinality of the dictionary may have changed. Depending on the new cardinality of the dictionary, fewer or more bits may be required to represent the compressed values of the column $j$. The number of bits required to represent $|U_{M+D}^j|$ distinct values is calculated as $\lceil \log_2 \left(|U_{M+D}^j|\right) \rceil$.

This step has to be done only once for the current column $j$. Thus, the runtime complexity of this step is constant $\mathcal{O}(1)$.

(b) **Update compressed values of** $M^j$, i.e. map uncompressed values of main $M^j$ and delta $D^j$ to new compressed values using the new dictionary $U_{M+D}^j$. This creates a new compressed main $M'^j$.

The following steps are required to create the new compressed main $M'^j$.

i. Iterate across the compressed values of the old main $M^j$. For each compressed value $K_C^{ij}$:
   A. **lookup** the old uncompressed representation $K^{ij}$ using the old dictionary $U_M^j$.
   B. **search** the new compressed representation $K_C'^{ij}$ of $K^{ij}$ using the new dictionary $U_{M+D}^j$.
   C. **append** the new compressed value $K_C'^{ij}$ to the new main storage $M'^j$.

ii. Iterate across the uncompressed values of the delta $D^j$. For each uncompressed value (1) *search* the compressed value using the new dictionary $U_{M+D}^j$ and (2) append it to the new main storage.

Please note that in the above enumeration, *lookup* denotes a linear search, whereas *search* indicates a binary search. Thus, both operations have different costs.

Since each uncompressed representation $K^{ij}$ is looked up in the dictionary using linear search the run-time complexity of this uncompression is $\mathcal{O}(|M^j|)$.

The search for compressed representations on the new dictionary $U_{M+D}^j$, however, is done using binary search due to the underlying CSB+ tree structure. Thus, the run-time complexity of both searches equals to

$$\mathcal{O}(|M^j| \cdot log(|U_{M+D}^j|) + |D^j| \cdot log(|U_{M+D}^j|))$$

The overall run-time complexity of the step is, therefore,

$$\mathcal{O}(|M^j| + (|M^j| + |D^j|) \cdot log(|U_{M+D}^j|))$$

Using the current algorithm, the attribute merge phase is dominated by the last step. This is confirmed by experiments, see figure 2.

As shown in figure 2, the overall run-time of the merge process is dominated by step 2 (b). As an optimization, we introduce two auxiliary data structures $X_M$, $X_D$ that correspond with the dictionaries of main and delta. For each dictionary entry, they store the index in the new dictionary $U_{M+D}^j$. They are created during the dictionary merge and can be used to reduce the binary search in step 2 (b) to a simple lookup.

## 3. GPU MERGE

As described in section 2 and illustrated in figure 2, the run-time costs of step 1 of the merge algorithm increases with the percentage of unique values. It is unlikely that the performance of step 1 (a) can be improved since the unique values are determined by a single iteration across the leaves of the underlying CSB+ tree structure. In general, the run-time of the merge algorithm is dominated by step 2 (b). However, due to the sequential dependencies in the algorithm, it cannot benefit from parallelization. In addition, it does not contain any computation or branching and is already bandwidth bound. The dictionary merge during step 1 (b), on the other hand, can be split up into smaller work items that can be processed independently. Thus, it is well suited for parallelization.

There are two reasons for researching the feasibility of a GPU implementation. Firstly, a GPU offers two orders of magnitude more cores than a CPU. This increases the maximum possible speedup through parallelization accordingly, as described by Amdahl's law. Secondly, the input data has to be copied before and the output data has to be copied after the computation on the GPU over the low bandwidth

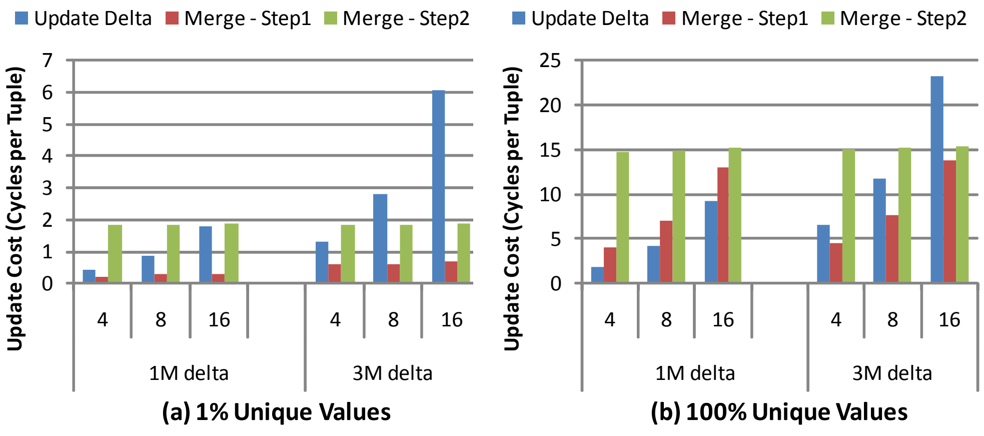


**Figure 2: Number of CPU cycles required for individual steps of merge process for different uncompressed value-lengths ($b^j$), varying delta sizes, and fractions of unique values. The size of the main storage is fixed to 100 m elements with 300 columns.**

PCI-Express bus. Thus, the actual performance improvement of the dictionary merge has to be large enough so that the overall processing on the GPU is faster than the execution on the CPU including the data transfer. Using the SHOC[1] benchmark suite, we measured a maximum write throughput to the GPU of 6 GB/s and a peak read throughput of 4.25 GB/s on our server used for benchmarking. Enabling ECC on the Tesla C2050 cards had no negative effect on the peak throughput rates.

In the remainder of this section, we will propose two parallel dictionary merge and one duplicate removal approach. In the following section 4, we will evaluate how well these approaches cope with the outlined requirements.

## 3.1 Parallel Dictionary Slice Merge

Parallel merge algorithms have been researched in great detail [28]. For instance, Joseph Jájá describes a parallel merge algorithm with a run-time complexity of $\mathcal{O}(\log\log n)$ [11].

The ideas of those parallel merge algorithms can be applied to step 1 (b) of the merge process - the merging of the main and delta dictionaries. However, traditional merge approaches have to be extended such that the auxiliary data structures are created while merging. In the following, we propose a parallel merge implementation that creates the auxiliary data structures $X_D$, $X_M$ that are required for updating the compressed values in step 2 (b).

### *Simplified Parallel Merge Strategy*

Due to the characteristics of the dictionaries that will be merged in step 1 (b), we can rely on some assumptions that simplify the design of our implementation. By definition, both dictionaries contain only sorted distinct values.

We define the term slice $s^d_{t_{ID}}$ as a subset of the dictionary $d$ containing the elements at the indices $k \in \{start^d_{t_{ID}}, \ldots, end^d_{t_{ID}}\}$. The notation $s^d_{t_{ID}}$ denotes that the slice of dictionary $d$ is merged by thread $t_{ID}$.

$$s^d_{t_{ID}} = \{v_k \mid k \in \{start^d_{t_{ID}}, \ldots, end^d_{t_{ID}}\}\}$$

Since the elements in each slice are ordered and distinct, all elements of a slice are smaller than the elements of the subsequent slice of the same dictionary $d$.

$$\forall v \in s^d_{t_{ID}} : \forall v_2 \in s^d_{t_{ID}+1} : v < v_2$$

This ordering property of the slices within one dictionary allows us to parallelize the merge. Since the output of the dictionary merge step is a sorted list of distinct values, the values in the slices will not change their position.

For simplification, we assume that the values in the slices of both dictionaries are ordered as well.

$$\forall v^1 \in s^1_{t_{ID}} \cup s^2_{t_{ID}} : \forall v^2 \in s^1_{t_{ID}+1} \cup s^2_{t_{ID}+1} : v^1 < v^2$$

Following this additional assumption, we provide the general idea of our first parallel merge strategy. It consists of three stages. While all stages can run in parallel without communication, there is an explicit barrier required between the second and third stage.

**(1) Initialization:** Each thread initializes its own local state required to start merging. Besides initializing its start and end indices, each thread locally initializes the `count` array. This avoids the serial execution of an initialization loop for the array. In addition, it reduces the initialization overhead of our future GPU implementation to the allocation of CPU and GPU memory as well.

**(2) Merge dictionary slices:** This stage of the algorithm mimics the behavior of the dictionary merge step 1 (b) already described in section 2.2. At the beginning of this stage, each thread points at the beginning of his slices $s^1_{t_{ID}}$, $s^2_{t_{ID}}$ using the indices calculated in the previous step $(i, k)$. The values $v^1_i$, $v^2_k$ are compared. The smaller value $v^d$ is inserted into the intermediate dictionary at index $w$. $w$ is then stored in the auxiliary structure of the dictionary $d$ at index $i$ if $v^1_i < v^2_k$ or at index $k$ if $v^1_i > v^2_k$. Each time both values in the dictionary match ($v^1_i == v^2_k$), $w$ is inserted in both auxiliary structures. In addition to comparing the values stored in both dictionaries, the number of iterations are counted as we can assume that they are equal to the number of unique values in both dictionaries.

This phase is ended by an explicit barrier to ensure that all threads have finally determined their local unique value counter.

[1]See `http://ft.ornl.gov/doku/shoc/start`.

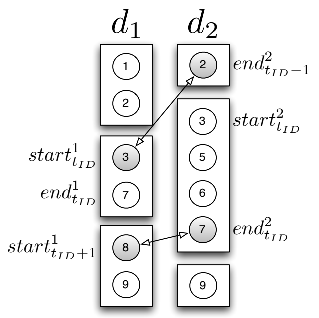


**Figure 3: Calculation of indices in $D_2$. The arrows indicate order between elements and their occurrence.**

**(3) Post-processing:** The post-processing writes the local results of the intermediate dictionary into the new dictionary. This is required, because there is the potential of gaps in the intermediate array since the number of distinct values per slice is unknown a priori.

The CPU allocates an array containing $|D_1| + |D_2|$ fields for the intermediate dictionary. Each thread can write into a segment of this array with $\frac{|D_1|+|D_2|}{threadCount}$ fields. All slices are equally sized but may have different numbers of unique values. Each thread that counts fewer than $\frac{|D_1|+|D_2|}{threadCount}$ distinct values, will leave a gap at the end of its section in the intermediate dictionary in the sense that it will only write as much values as it found distinct values. The number of distinct values may range between $max(|s^1_{t_{ID}}|, |s^2_{t_{ID}}|)$ and $|s^1_{t_{ID}}| + |s^2_{t_{ID}}|$.

As well as the dictionary merge step 1 (b) of the merge process, the runtime complexity of this parallel merge strategy is driven by the size of both dictionaries $\mathcal{O}(|D_1|+|D_2|)$.

### *Generalized Parallel Merge Strategy*

While the general idea described for the simplified parallel merge is valid, it relies on an unrealistic assumption concerning the data characteristics of the slices.

$$\forall v^1 \in s^1_{t_{ID}} \cup s^2_{t_{ID}} : \forall v^2 \in s^1_{t_{ID}+1} \cup s^2_{t_{ID}+1} : v^1 < v^2$$

In general, this ordering property cannot be guaranteed in the simplified parallel merge strategy since the second list is cut into equally sized slices. While the first dictionary can still be cut into equally sized slices, the boundaries of the second slice have to be determined depending on its actual values. The predecessor of the start element $start^2_{t_{ID}}$ has to be the last element in $d_2$ smaller than the start element of the first slice $start^1_{t_{ID}}$. Similarly, the last element of the slice $s^2_{t_{ID}}$ is the last element in $d_2$ which is still smaller than the successor of $end^1_{t_{ID}}$, see figure 3.

$$\begin{aligned} D_2[start^2_{t_{ID}} - 1] &< D_1[start^1_{t_{ID}}] \\ D_2[end^2_{t_{ID}}] &< D_1[end^1_{t_{ID}} + 1] \\ \nexists i > start^2_{t_{ID}} &: D_2[i-1] < D_1[start^1_{t_{ID}}] \\ \nexists i > end^2_{t_{ID}} &: D_2[i] < D_1[end^1_{t_{ID}}] \end{aligned}$$

Since both dictionaries are sorted, we apply binary search

| Publication | Author | Device | Rate CPU | Rate GPU |
|---|---|---|---|---|
| 2009 [23] | NVIDIA | GTX 280 | | 200 |
| 2010 [25] | Intel | Knights Ferry vs. GTX 280 | **560** | 176 |
| 2010 [24] | Intel | Core i7 vs. GTX 280 | 250 | 200 |
| 2009 [13] | University | Tesla C1060 | | 300 |
| 2010 [14] | University | GTX 285 | | 550 |
| 2011 [15] | University | GTX 480 | | **1005** |

**Table 2: Comparison of different reported sort implementations on CPUs and GPUs. The rates describe the number of random 32-bit keys sorted per second, given in $10^6$ keys per second.**

to determine the indices $start^2_{t_{ID}}$, $end^2_{t_{ID}}$ with a run-time complexity of $\mathcal{O}(\log(|D_2|))$.

During the initialization step, each thread looks up its start and stop indices and merges the dictionaries as described for the simplified merge strategy. When each thread computes both indices individually, no synchronization between the threads is required during the initialization. However, it is sufficient to let each thread calculate only its start index $start^2_{t_{ID}}$. It can then use the predecessor of the start index of the next thread as its end index.

$$end^2_{t_{ID}} = start^2_{t_{ID}+1} - 1$$

Alternatively, each thread can calculate its end index $end^2_{t_{ID}}$ and then conclude its start index as follows:

$$start^2_{t_{ID}} = end^2_{t_{ID}-1} + 1$$

This approach, however, requires an explicit barrier after the calculation of the first index. This barrier has to make sure that all threads have determined their respective indices and stored them into shared memory accessible by the other threads. Without this barrier, threads may conclude their second index inconsistently, e.g. by reading the index of the succeeding or preceding thread from uninitialized or outdated shared memory due to the caching of global memory.

Furthermore, the barrier has to guarantee to each thread that it can read the index calculated by its successor or predecessor from shared memory. Since threads are organized into blocks in the CUDA programming model, a block wise synchronization, e.g. `__syncthreads()`, is not sufficient. If thread $t_{ID}$ and $t_{ID}+1$ reside on different thread blocks, both can pass their synchronization barrier even though they cannot yet see the index calculated by the thread they depend on. Thus, this design requires a synchronization primitive that guarantees consistency across blocks [6, 32].

The calculation of the boundaries of the slices increases the runtime complexity to $\mathcal{O}(|D_1| + |D_2| + \log|D_2|)$.

## 3.2 Alternative Parallel Merge Algorithms

We want to reuse mature existing parallel programming libraries that are optimized for todays GPUs. Therefore, we map the problem of parallel merging to the building blocks provided by those libraries. We expect such an implementation may perform better with respect to runtime. **Thrust** is a is a library for NVIDIA CUDA applications that mimics the STL, see `http://code.google.com/p/thrust/`. It provides algorithms and data structures, that are typically required for writing parallel applications. With respect to the algorithms provided by the library, we propose two parallel merge algorithms.

### *Concatenate-Sort-Unique-Map (CSUM)*

The CSUM approach consists of four stages. Each stage uses a different primitive provided by Thrust. Firstly, both dictionaries are concatenated to create a single list with $|D_1|+|D_2|$ values. This is done while copying the input data to the GPU's memory using the `copy` function. Secondly, the new concatenated list is sorted on the GPU. Sorting is a well researched topic with a large number of publications providing implementations for modern CPUs and GPUs [23, 13, 25, 24, 14, 15]. Measurements of maximum sorting rates differ across publications. Thus, table 2 provides an overview of the latest maximum sort rates on CPUs and GPUs for uniformly distributed random 32-bit keys. The table summarizes the best sort rates that could be achieved by the authors. Thirdly, the duplicates are removed from the sorted list to create the new sorted dictionary that contains only distinct values using the `unique` primitive. Fourthly, each value of the input dictionaries is mapped to its position in the new dictionary using either `transform` with binary search or `lower_bound`.

### *Merge-Unique-Map (MUM)*

Since the CSUM approach does not exhibit the fact that both input data structures are already sorted, we propose the MUM strategy which takes the data characteristics into account. It consists of three stages. Firstly, both input data structures are merged to create a single sorted sequence using the `thrust::merge` primitive. This step produces the same output as the concatenate and sort stages of the CSUM approach. As a consequence, the MUM strategy uses the `unique` and `lower_bound` operations during the last two steps as described for the CSUM approach.

### *Reduced Development Time*

As described by Foster, the development costs have to be considered as well when designing and building parallel systems [7]. Therefore, we expected that the reuse of the optimized GPU primitives would result in reduced development time and better performance. Compared to the dictionary slice merge, the implementation of the alternative merge approach took less than half the time when we used the Thrust library. This enabled us to concentrate on measuring and improving performance.

## 3.3 Duplicate Removal

HYRISE uses a CSB+ tree to compress the data stored in the delta. While this increases the insert costs, it makes it cheap to identify the unique elements of the delta in step 1 (a), see section 2. In update intensive workload, the insertion cost into the CSB+ tree may be too high. In this case, it may be cheaper just to insert the data without applying compression. While such a design would support high update rates, it would be more expensive to build a dictionary for the delta. When no CSB+ tree is used to store the values of the delta, a fast duplicate removal implementation must be used during the merge instead of iterating across the leaves of the tree.

Instead of implementing our own duplicate removal approach, we used primitives of the Thrust library. We used `thrust::sort` and `thrust::unique` and compared their performance to their STL counterparts `std::sort` and `std::unique`.

## 4. EVALUATION

We benchmarked the CPU and GPU implementations on an idle Linux server with an eight core Intel Xeon E5620 processor running at 2.4 GHz, 24 GB of main memory, and four Tesla C2050 cards. The CPU implementation ran on one CPU core while the GPU implementation was executed with different numbers of blocks and threads on a single Tesla C2050 GPU. The dictionaries contained four byte integer values. All examples have been compiled using `g++` 4.4 with the `-O3` flag. The GPU implementations have been compiled with `nvcc` 4 and run using the CUDA 4 driver.

Our dictionary slice merge implementation for GPUs does not use any synchronization between the threads. Each thread individually calculates the boundaries of its slice $s^2_{t_{ID}}$ and merges it with $s^1_{t_{ID}}$ as described in the previous section. While the avoidance of synchronization removes the respective runtime overhead, it forces us to apply a significant amount of post processing on the CPU after running the kernel. Thus, parallelism is only achieved during the initialization and the merge phase. The current implementation post processes the data sequentially on the CPU. The GPU implementation achieved about half the throughput rate of the CPU implementation. While the performance differs depending on the number of blocks and threads per block, the GPU implementation reaches its peak throughput when using between 64x64 and 256x256 threads.

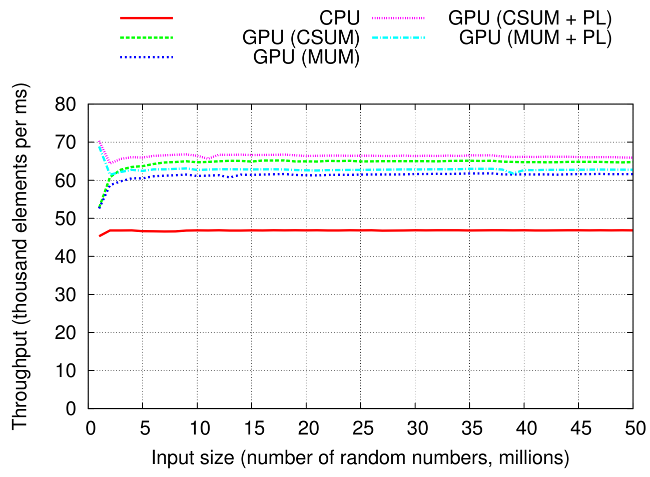
The characteristics of the current algorithm make it difficult to exploit the full computational power of the GPU hardware. Depending on the input data, there is the potential for high branch divergence rates. During each iteration of the main merge loop, four conditions are checked, four integer values stored in registers are incremented, and two global memory locations are written to. In addition, the binary search for the start and end indices of the slice $s^2_{t_{ID}}$ requires at most $3 \cdot \log(|D_2|)$ condition checks and integer additions. This relatively high probability of branch divergence between the threads coupled with the low arithmetic intensity is responsible for the limited merge performance.

Further measurements with **cudaprof** reveal that the parallel merge on the GPU is dominated by copying the memory between main memory and the device. Depending on the input data size, the runtime of the kernel accounts for less than 30% of the overall merge time.

The throughput of the alternative parallel dictionary merge, was up to 40% higher than the CPU implementation. In both GPU merge implementations, more than 60% of the run-time are spent for data transfers.

Depending on the input size, the throughput of our GPU duplicate removal implementation was up to 27 times faster than the single threaded STL implementation.

We decided to compare the performance of a single GPU with a single CPU core. Because when we merge database tables we want to merge one column per core. Thus, in the current 8 core system, 8 columns can be merged in parallel by CPU GPU pairs. In this case, we are interested in the performance of a single core rather than multiple cooperating cores. When work is scheduled differently among the compute resources, the measured performance of a single CPU core can be multiplied to simulate perfect scalability using an arbitrary number of cores.



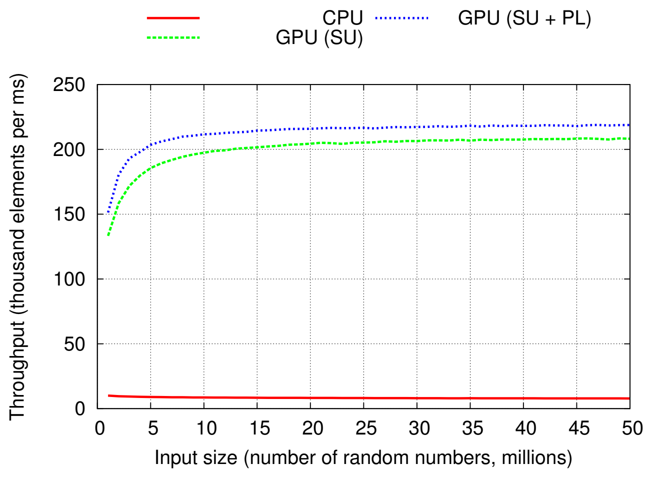


**Figure 4: Left: Throughput of the CSUM and MUM parallel GPU merge against the CPU (STL) implementation. Right: Duplicate removal throughput of GPU (with Thrust) vs. CPU (with STL). The graphs with the "+ PL" suffix show the GPU throughput of the respective implementations when page-locked memory is used.**

## 5. FUTURE WORK

So far, we described the algorithm only for merging numbers. To support merging of the main dictionary and the unique values of the delta, the parallel merge algorithm needs to be able to merge strings. We propose two different strategies for supporting string merging. Firstly, the strings can be temporarily mapped to digits for the duration of the merge. Thus, the strings are mapped to digits before running the merge through hashing. Then, the algorithm can be used as described above. After the merge, the values in the resulting dictionary have to be mapped back to strings again. The data in the auxiliary structures has not to be treated differently, since it only stores the positions of the respective values in the new dictionary. Secondly, the merge algorithm can be designed to operate directly on strings. Instead of (un-)mapping the strings before and after the actual merge process, it compares the strings directly instead of comparing their hashcode equivalents.

Furthermore, the cost of memory transfer between main-memory and GPU memory needs to be evaluated in greater detail. While the current implementation is straight-forward. It is indispensable to think about the possibility to schedule work on the GPU and try to the hide memory transfer latency. The full potential for offloading the merge to the GPU can only be unleashed when the cost for memory transfer are reduced.

Another question is, in which scenarios it is worthwhile to abandon the CSB+ tree in the delta to support faster modifications by storing all data in uncompressed form. In this case, the fast duplicate removal implementation can be used so that the overall merge run-time stays the same while the cost for inserts is reduced. However, it has to be taken into account that not compressing the delta reduces the read performance on it.

## 6. RELATED WORK

Severance and Lohman describe an architecture that relies on a "differential database representation" which stores database modifications separately [27]. It allows the last snapshot of the database, the reference point, to be stored efficiently. In addition, it facilitates fast and inexpensive backup and recovery since all modifications are stored at a single place.

O'Neil et al. propose the log-structured merge (LSM) tree data structure in scenarios with stringent requirements for read performance and high update rates [18]. Therefore, the LSM tree defers updates through buffering and applies them in batches.

To support fast updates in read-mostly applications, e.g. customer relationship applications, **C-Store** distinguishes between a compressed *read-optimized store* (RS) and an uncompressed *writeable store* (WS) [30]. As a background task, the *tuple mover* transfers updates and inserts from the WS into the RS. A new read-optimized storage $RS'$ is created, where deleted items are skipped, i.e. discarded from $RS$ so that they do not appear in $RS'$, updates and inserts are migrated to $RS'$. This is called the *merge out process* (MOP). To avoid lock contention in workloads with small numbers of OLTP transactions, C-Store uses *snapshot isolation* instead of conventional locking.

**MonetDB/X100** does not describe a merge process explicitly [33, 2]. There is, however, the notion of delta columns for storing inserts. Updates are implemented as a delete followed by an insert. Deletes are implemented by appending the tuple identifier to a deletion list. MonetDB will reorganize the data every time the size of the delta columns exceeds a small percentile of the total table size.

Besides GPUs, there are other accelerators [16] designed for supporting data parallelism which we do not consider in this paper. For instance, the Cell broadband engine [3, 10], IBMs Power EN processor [21], and Intel's upcoming Many Integrated Core platform [25].

The problem of partitioning the data and scheduling the tasks across the compute units is critical to the overall system performance [7]. Augonnet et al. describe an elegant approach of automatic partitioning of work and self-adjusting scheduling across heterogeneous compute units in their StarPU project [1].

## 7. CONCLUSION

In this paper, we analyzed the feasibility of a parallel merge implementation for GPUs. We found that there is only a limited amount of performance improvement achievable through parallelization since only the first step of the merge is likely to benefit from the level of parallelization offered by GPUs. For instance, the overall merge algorithm can be accelerated by a factor of two approximately when all values are distinct, the delta contains three million items, and the uncompressed value length $b^j$ is 16 bytes, see figure 2. The time of the merge phases depends on characteristics of data, e.g. number of unique values and uncompressed value length $b^j$. Only for large numbers of unique values, the dictionary merge phase is relatively expensive. In these cases, the overall run-time of the merge processes can be shortened by parallelizing the first step.

While the throughput of our GPU duplicate removal implementation was 21-27 times higher than the CPU implementation, the actual dictionary merge 1 (b) could only be improved by about 40%. Our dictionary slice implementation achieved about half the throughput of the CPU. In both GPU merge implementations more than 60% of the time are spent for data transfers. While the duplicate removal implementation demonstrates the speedup that can be achieved without much effort, further research is required to identify and circumvent the performance bottleneck of the current GPU dictionary merge implementations - the data transfers between main memory and the GPU.

From our own experience, we recommend using a parallel programming library that enables rapid prototyping, testing, and early performance optimizations. While the ideas behind our dictionary slice algorithm may be well suited for parallelization, non-optimal memory access patterns and synchronization problems of the current implementation result in bad performance compared to the other implementations that we built using the Thrust library.

## 8. REFERENCES

[1] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA, Mar. 2010.

[2] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[3] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation - A performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[4] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6):85–98, 1992.

[5] W. Fang, B. He, and Q. Luo. Database Compression on Graphics Processors. *VLDB*, 2010.

[6] W. Feng and S. Xiao. To GPU synchronize or not GPU synchronize? In *ISCAS*, 2010.

[7] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley, 1995.

[8] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *COMAD*, 2006.

[9] M. Grund, J. Krueger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 2010.

[10] C. Johns and D. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[11] J. Joseph. An Introduction to Parallel Algorithms, 1992.

[12] J. Krüger, M. Grund, J. Wust, A. Zeier, and H. Plattner. Merging Differential Updates in In-Memory Column Store. In *DBKDA*, 2011.

[13] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *IPDPS*, 2009.

[14] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *PACT*, 2010.

[15] D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *PPL*, 2011.

[16] A. Munshi et al. The OpenCL specification version 1.0. *Khronos OpenCL Working Group*, 2009.

[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[18] E. Patrick, E. Cheng, D. Gawlick, and J. Elizabeth. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf*, 33(4):351–385, 1996.

[19] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[20] J. Rao and K. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, volume 29, 2000.

[21] G. Risi. PowerEN - An "Edge of Network Processor". Technical report, IBM Systems and Technology Group, August 2010.

[22] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[23] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, 2009.

[24] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *COMAD*, 2010.

[25] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs, GPUs and Intel MIC Architectures. Technical report, Intel, 2010.

[26] M. Schatz and C. Trapnell. Fast exact string matching on the GPU. *CBCB*, 2007.

[27] D. Severance and G. Lohman. Differential files: their application to the maintenance of large databases. *TODS*, 1(3):256–267, 1976.

[28] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[29] T. Siriwardena and D. Ranasinghe. Global Sequence Alignment using CUDA compatible multi-core GPU. 2009.

[30] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[31] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[32] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, 2010.

[33] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS in the CPU cache. *IEEE Data Engineering*, 2005.