Comparison Based Sorting for Systems with Multiple GPUs

Ivan Tanasic[§], Lluís Vilanova[§], Marc Jordà[§], Javier Cabezas[§], Isaac Gelado[§]

Nacho Navarro[§‡], Wen-mei Hwu[*]

[§]Barcelona Supercomputing Center {first.last}@bsc.es
[‡]Universitat Politecnica de Catalunya nacho@ac.upc.edu
[*]University of Illinois w-hwu@illinois.edu

ABSTRACT

As a basic building block of many applications, sorting algorithms that efficiently run on modern machines are key for the performance of these applications. With the recent shift to using GPUs for general purpose compuing, researches have proposed several sorting algorithms for single-GPU systems. However, some workstations and HPC systems have multiple GPUs, and applications running on them are designed to use all available GPUs in the system.

In this paper we present a high performance multi-GPU merge sort algorithm that solves the problem of sorting data distributed across several GPUs. Our merge sort algorithm first sorts the data on each GPU using an existing single-GPU sorting algorithm. Then, a series of merge steps produce a globally sorted array distributed across all the GPUs in the system. This merge phase is enabled by a novel pivot selection algorithm that ensures that merge steps always distribute data evenly among all GPUs. We also present the implementation of our sorting algorithm in CUDA, as well as a novel inter-GPU communication technique that enables this pivot selection algorithm. Experimental results show that an efficient implementation of our algorithm achieves a speed up of 1.9x when running on two GPUs and 3.3x when running on four GPUs, compared to sorting on a single GPU. At the same time, it is able to sort two and four times more records, compared to sorting on one GPU.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Algorithms, Design, Performance

Keywords

Parallel, Sorting, GPU, CUDA

GPGPU-6, March 16 2013, Houston, TX, USA

1. INTRODUCTION

Sorting is a key building block in many High Performance Computing (HPC) applications. Examples of these are N-body simulations [1], some high performance sparse matrix-vector multiplication implementations [2], graphics algorithms like Bounding Volume Hierarchy (BVH) construction [3], database operations [4], machine learning algorithms [5] and MapReduce framework implementations [6, 7].

A common trend in computing today is the utilization of Graphical Processing Units (GPUs) that efficiently execute codes rich in data parallelism, to form high performance heterogeneous systems [8]. In these CPU/GPU systems, application phases rich in data parallelism are executed in the GPU, while the remaining portions of the application are executed on the CPU. However, depending on the phase of the application, this pattern requires moving the data back and forth between the CPU and the GPU, introducing overheads (in both execution time and power consumption) that can void the benefit of using GPUs. To avoid these overheads in GPU accelerated applications that use sorting, several sorting algorithms that execute in the GPU have already been proposed, including [9, 10, 11, 12, 13]. Most of the existing GPU sorting algorithms only support single-GPU systems. This limitation might be unacceptable in applications that make use of several GPUs for a number of reasons. It would lead to underutilization of the available resources, increased overhead of moving the data to one GPU to be sorted and, since data structures might be distributed across the memory of all GPUs in the node, their total size can exceed the memory capacity of a single GPU. To the best of our knowledge, only the algorithms presented in [9, 14, 15] allow using more than one GPU, but these algorithms are either designed for external (out-of-core) sorting or have limitations.

In this paper we present an efficient multi-GPU internal (in-core) merge based sorting algorithm that allows sorting data structures already distributed across the memories of several GPUs, including very large structures that could not fit in a single GPU and runs on any number of GPUs connected to the same compute node. We do not target cluster level sorting, although our sorting algorithm could be used as a node level sort operation when building a cluster level sort. It consists of two phases: a first phase which performs a local sort in each GPU, resulting in "per GPU" sorted data, and a second phase which performs a series of merges over ordered data from different GPUs. We designed the

merge step to be able to cope with the distributed nature of a multi-GPU system, utilize the memory perfectly and keep the system in complete load balance. This is achieved by splitting the arrays to be merged in two parts in such a way that exchanging exactly the same number of elements between them as two contiguous chunks of memory would results in the data that can be merged by each GPU to obtain the globally sorted array. Here we introduce a novel pivot selection algorithm that guarantees these properties.

This paper also presents the CUDA implementation of our proposed multi-GPU merge sort algorithm. Our implementation uses peer-to-peer (P2P) GPU access, introduced in CUDA 4.0, to enable efficient inter-GPU communication, which is key to accomplish a high performance merge step. However, P2P GPU access is only available for GPUs connected to the same PCI express (PCIe) controller. To enable low-overhead inter-GPU communication across GPUs connected to different PCIe controllers, we introduce a novel inter-GPU communication technique based on the host mapped memory mechanism [16].

The combination of the algorithm and the communication mechanisms (P2P GPU accesses and our inter-GPU communication technique) allows us to accomplish a scalable CUDA implementation of multi-GPU sorting that scales up to 1.9x when sorting on two GPUs, and up to 3.3x when sorting on four GPUs, compared to the single-GPU sorting. Furthermore, implementing a fundamental algorithm such is sorting for a multi-GPU system allows us to evaluate this emerging platform and the mechanisms it offers.

The paper is organized as follows: Section 2 presents the necessary background material on sorting algorithms and the base system we assume in this paper; Section 3 provides an overview of the algorithm, while Section 4 goes through all those insights that actually provide an efficient implementation; Section 5 discusses the experimental evaluation of the proposed solution; Section 6 presents all the relevant work related to sorting on GPUs; finally, Section 7 draws the final conclusions of this work.

2. BACKGROUND AND MOTIVATION

This section presents the necessary background information that is required in order to understand the following sections. We first describe the families of sorting algorithms and then we describe our target system.

2.1 Sorting Algorithms

Sorting algorithms can be divided into two families depending on their behavior: comparison and non-comparison based sort. Comparison based sorting algorithms traverse a list of elements performing an arbitrary comparison operation that can work on any pair of elements. Thus, the comparison operator defines the ordering (e.g., ascending) that the resulting list of elements must comply with. This family includes well-known algorithms such as Quicksort [17] and Merge sort [18]. By contrast, non-comparison based sorting algorithms can achieve higher performance by fixing the potential comparison operators, and thus limiting the input data types and output ordering (e.g., integer ascending). This family includes algorithms such as Bucket sort and Radix sort [18]. In this paper we implement a comparison based sorting algorithm, merge sort, because the ability to define custom comparison operators expands the number of applications that can potentially benefit of this work.

Most parallel versions of comparison-based sorting algorithms are based on two approaches. *Merge-based* approach splits data into chunks, sorts these chunks independently (using some "small sort" algorithm) and then performs a number of merge steps to obtain globally sorted data. The other approach, *distribution-based*, sets up a number of buckets, for each element computes the destination bucket, scatters all elements to the right buckets, sorts these buckets independently, and finally concatenates all buckets to get the globally sorted data. Buckets are usually set up through sampling (deterministic or random, depending on the approach) in an effort to balance their occupancy. Picked up samples represent the bucket range, while some form of histogramming is usually performed to determine the size of each bucket.

2.2 Algorithmic Approach

In this paper we target a merge-based algorithm which paired with our pivot selection algorithm allows us to build a sorting mechanism suitable for the distributed structure of a multi-GPU system. This is achieved by moving the data in chunks to it's destination GPU in a series of merge steps and finally performing the merge locally, in each GPU.

Compared to typical distribution-based sorting mechanisms, our approach to merge sort provides several benefits. It is able to efficiently utilize the available memory since there are no data marshalling stages performed and all the data exchanged by pairs of GPUs is of the same size, so it can fit in the place of the data that it is being exchanged with. As a result of this, our algorithm ensures that each data partition fits in the GPU and is able to keep the system in complete load balance. Pivot selection is the only part of the algorithm that could benefit from a shared memory architecture and, as we show later, modern GPUs are able to cope with this.

Distribution-based sorting algorithms are, on the other hand, optimal regarding the communication (necessary number of transfers of a single element until it reaches the destination processor) and hence are the preferred approach for sorting on big clusters and supercomputers. Because our targeted systems, described below, are small scale systems, communication overhead is less of a concern. This is also shown through the experimental evaluation in Section 5.

2.3 Target System

Even though GPUs are best suited for executing compute intensive, data parallel algorithms with regular memory access, the large number of in-flight threads and high memory bandwidth also enable some non-compute intensive operations like database [19], data mining [5] and map-reduce [6] primitives to achieve high performance when executed on a GPU. Sorting is not a compute-intensive task, but rather a combinatorial problem. However, researchers have shown a number of GPU sorting algorithms that are competitive to, or faster than, parallel implementations of sorting on CPUs. This means that GPU applications that rely on the sorting operation can execute faster than they would if they were offloading sorting to the CPU.

Recently, integrated CPU/GPU systems that share the same physical memory, such as AMD Fusion [20] chips, have been introduced. The rationale behind this integration is to avoid copies among CPU and GPU memories to improve their communication latency, which is an important factor

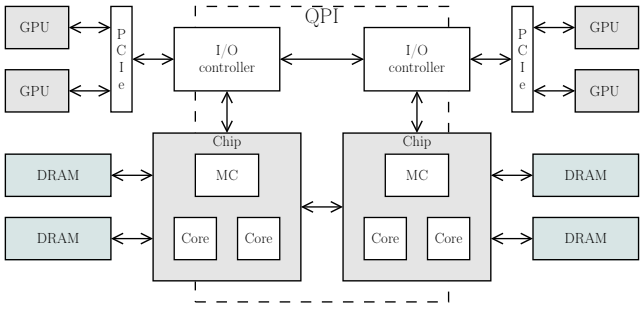


Figure 1: Physical organization of the target system.

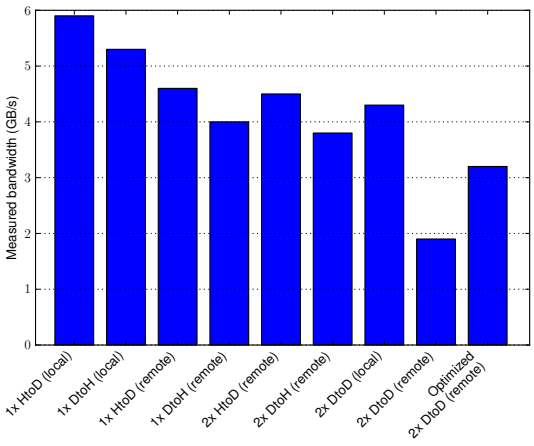


Figure 2: Measured peak bandwidths (H is host, D is device) when performing a single memory transfer (1x) or two concurrent memory transfers on different devices (2x). Bandwidths for concurrent transfers are per device (not aggregated). Local transfers are inside the same NUMA domain, remote transfers are between two NUMA domains.

in many gaming and multimedia applications. However, the different needs of CPUs and GPUs on the memory subsystem (e.g., bandwidth versus latency) make integrated GPUs deliver a fraction of the performance that discrete GPUs are currently offering. Thus, this scenario is reserved mainly for low power and mobile devices, while most workstations and HPC systems attach one or several GPU cards to the system via I/O extension slots on a PCI express (PCIe) bus. In this paper, we target systems with more than one GPU attached via PCIe buses.

A typical multi-GPU system is shown in Figure 1. It contains multiple CPUs, connected to DRAM memory via on-chip memory controllers, thus creating a non-uniform memory access (NUMA) system. CPUs are connected to one or more PCIe buses, via I/O controllers, to which there are one or more GPUs attached. All the CPUs are connected to each other with a full mesh interconnect. In the case of Intel systems, the network is Intel QuickPath^{tm} Interconnect [21] (QPI), while in the case of AMD systems, CPUs are interconnected with a similar interconnection network called Hypertransport^{tm} [22]. Currently, multi-GPU machines have up to eight GPUs, with most machines having four or fewer. The number of GPUs in a single machine is not expected to grow much in the near feature, for similar reasons as the number of CPU sockets in a node is not growing (e.g., data sharing issues).

Modern GPUs (NVIDIA from CUDA 4.0 and those that will support HSA [23]) have a unified virtual address space (UVAS) [24] for all the GPUs and CPUs in the system. The UVAS allows GPUs to transparently use P2P communication, if the underlying interconnect permits it, allowing host-initiated transfers of data between two GPUs (*P2P memory transfer*). This allows a more efficient inter-GPU communication than in previous GPU generations which required copying the data from one GPU to host memory, and then from host memory to the other GPU. UVAS and P2P communication also let the GPU transparently issue load and store instructions that directly read and write device memory of another GPU (*P2P memory access*). In our implementation we use P2P memory transfers during the merge steps, and P2P memory accesses in our pivot selection algorithm, as described in Section 4.

Both PCIe and current CPU interconnects support P2P communication, but communication of peers on different PCIe buses is not possible. This means that P2P transfers and P2P accesses are possible only between GPUs attached to the same PCIe bus. Today, most of the manufacturers of GPU systems design their machines with one or two GPUs per PCIe bus to prevent the bus from becoming a bottleneck, since the bandwidth of the PCIe bus is shared among all the GPUs on that bus. Most systems with four GPUs attach one pair of GPUs per PCIe bus, and thus P2P communication can only be used between pairs of GPUs. In these systems, P2P transfers between GPUs on different PCIe buses are emulated using intermediate host memory copies.

The NUMA nature of our target system has major implications on the performance of applications and thus it must be treated carefully [25]. In order to obtain maximum performance when transferring data between device and host, host memory needs to be allocated in the physical memory that is closer to the PCIe bus where the GPU is attached to (see Figure 1). Such a GPU is called "local" to the CPU closest to that physical host memory. Figure 2 shows the measured bandwidths on our test machine, which is described in more detail in Section 5.

The first set of experiments measure the bandwidth for a single transfer from the host (host mapped buffer) to a device (*HtoD*) and vice versa (*DtoH*). In the case of using a local device, the transfer is limited by the bandwidth of PCIe, while for the case of a remote device the available bandwidth decreases due to the hop transfer through the QPI interconnect. Being a full duplex bus, QPI is able to sustain the bandwidth while handling two concurrent transfers (see "*2x HtoD*").

The second set of experiments measures the bandwidth for two concurrent transfers from one device to another, which is one of the basic operations performed in this work. When GPUs are on the same PCIe bus (*DtoD local*), bandwidth is as expected. When GPUs do not share the same PCIe bus (*DtoD remote*), bandwidth in our system drops dramatically. As already stated, there is no P2P transfer support between two GPUs attached to different PCIe buses, meaning that the runtime needs to allocate intermediate host buffers, transfer the data from one GPU to host memory,

and then back from the host to the other GPU. Since this transfer operation is an important part of our proposed algorithm, we have implemented an optimized version that is NUMA-aware and performs double-buffering using *pinned* intermediate host buffers. As can be seen in Figure 2 (*DtoD remote optimized*), the achieved bandwidth is substantially higher than that of the default implementation provided by CUDA. This is however still significantly lower than what is expected (bandwidth close to the "*2x HtoD*"). We suspect that concurrent memory reads and memory writes necessary to implement double-buffering cause contention in the host memory controllers, preventing us to achieve expected bandwidth.

3. ALGORITHM OVERVIEW

This section presents and analyzes the two-phase multi-GPU merge sort algorithm along with its key enabler, the efficient pivot selection algorithm.

3.1 Algorithm Structure

Initially, the records to be sorted are evenly distributed among the memories of the GPUs in the system. The proposed multi-GPU merge sort algorithm consists of two main phases. First, each GPU independently sorts the records hosted in its memory. The second phase consists of one or several merge stages, depending on the number of GPUs. The choice of the single GPU sorting algorithm used in this first phase is independent of the second phase of the algorithm. In this section we discuss each of the phases.

3.2 Single-GPU Sorting Phase

Merge sort can be implemented starting from n arrays containing one single record and then performing $\log_2 n$ steps of pairwise merging (where n is the number of records to be sorted). However, this approach usually leads to inefficient implementations, so merge sort is often implemented as a hybrid sorting algorithm, consisting of two phases:

1. The input array is divided into multiple chunks of m records each. Then, an algorithm to efficiently sort the small number of records within each chunk is used. There are several suitable alternatives for the small sort stage, such as quicksort, a bitonic sorting network or an odd-even sorting network.

2. $\log_2 \frac{n}{m}$ pairwise merging stages are performed with $\frac{n}{m}$ already sorted arrays. After each merge stage, the number of sorted chunks is halved, and the size of each chunk is doubled.

A naive single-GPU merging implementation, which sequentially merges pairs of arrays, loses half of the available parallelism in each step, and thus is not suitable for execution on a massively parallel system such as a GPU. To avoid this problem, for a single-GPU sorting we use an implementation based on the parallel merge sort described in [10], taken from the Thrust library [26]. This algorithm uses an odd-even sorting network for its first phase. The second phase is based on finding the ranks of the elements. The rank of one element in an array is defined as the number of elements in the array that are smaller than the given element. For every record in each pair of arrays to be merged, the final position of the record is calculated as the sum of the two record's ranks (one for each array). The rank of one element in its originating array is trivially given by its position (index) in that array, while the rank in its pair array can be found via binary search for the given element in the pair array. Hence, the final positions of all the elements from both arrays can be found in parallel. To improve locality, the blocking mechanism proposed in [27] is also used in this single-GPU sorting algorithm.

3.3 Multi-GPU Merge Phase

Merging the data distributed among GPUs in our algorithm relies on a key observation (Observation 1). Given the two sorted arrays A_α and A_β, there exist a pivot point P in A_β and its "mirrored" counterpart P' in A_α that partition the input arrays into two parts, upper and lower, such that elements from both lower parts are smaller than or equal to the elements from both upper parts while the number of elements in the lower part of each array is equal to the number of elements in the upper part of the other array. Merging lower parts and merging upper parts will result in two sorted arrays which when concatenated provide one, sorted array.

OBSERVATION 1. *Given two sorted arrays A_α and A_β:*

$$\exists P \in \{0 \dots |A_\beta| - 1\}, P' \in \{0 \dots |A_\alpha| - 1\} \big| \\ P' = |A_\alpha| - P - 1$$

where P and P' partition A_α and A_β into:

$$A_{\alpha,lower} \equiv A_\alpha[0 \dots P'], A_{\alpha,upper} \equiv A_\alpha[P' + 1 \dots |A_\alpha| - 1], \\ A_{\beta,lower} \equiv A_\beta[0 \dots P - 1], A_{\beta,upper} \equiv A_\beta[P \dots |A_\beta| - 1]$$

such that:

$$A_{\alpha,lower} \leq A_{\beta,upper}, A_{\beta,lower} \leq A_{\alpha,upper}$$

It is important to note that in Observation 1 the relation between pivots P' and P guarantees that $A_{\alpha,upper}$ and $A_{\beta,lower}$ will have the same number of elements. This enables an efficient inter-GPU merge operation, where a simple swap of consecutive records ($A_{\alpha,upper}$ and $A_{\beta,lower}$) paired with a merge stage allows the merging of arrays distributed on several GPUs. Being able to just swap two parts of memory means that there are no additional data marshalling stages, new memory allocations, etc.. Furthermore, this algorithm allows keeping the data evenly distributed across GPUs and ensures that each data partition always fits in the GPU memory. Note that when multiple candidate pivots exist, the smaller pivot P is used in order to minimize the amount of data to swap. A more detailed descriptions of swap operation and the pivot selection algorithm are provided later.

Once the input records are sorted in each GPU (initial state of arrays $A_{0\dots3}$ in Figure 3), $\log_2 G$ merge stages are performed, where G is the number of GPUs being used. Each merge stage takes two sorted input arrays of m records, each array distributed across g GPUs, and produces a single sorted array of $2m$ records distributed across $2g$ GPUs. When each input array of a merge stage is spread on two or more GPUs, a series of intermediate merge steps need to be performed which, breaking the problem of merging into smaller merge steps, until they can finally be performed inside one GPU.

Figure 3 illustrates the algorithm for the case of four GPUs. The first stage only involves pairs of GPUs (steps 1 through 3 in Figure 3), so one pivot and its counterpart have

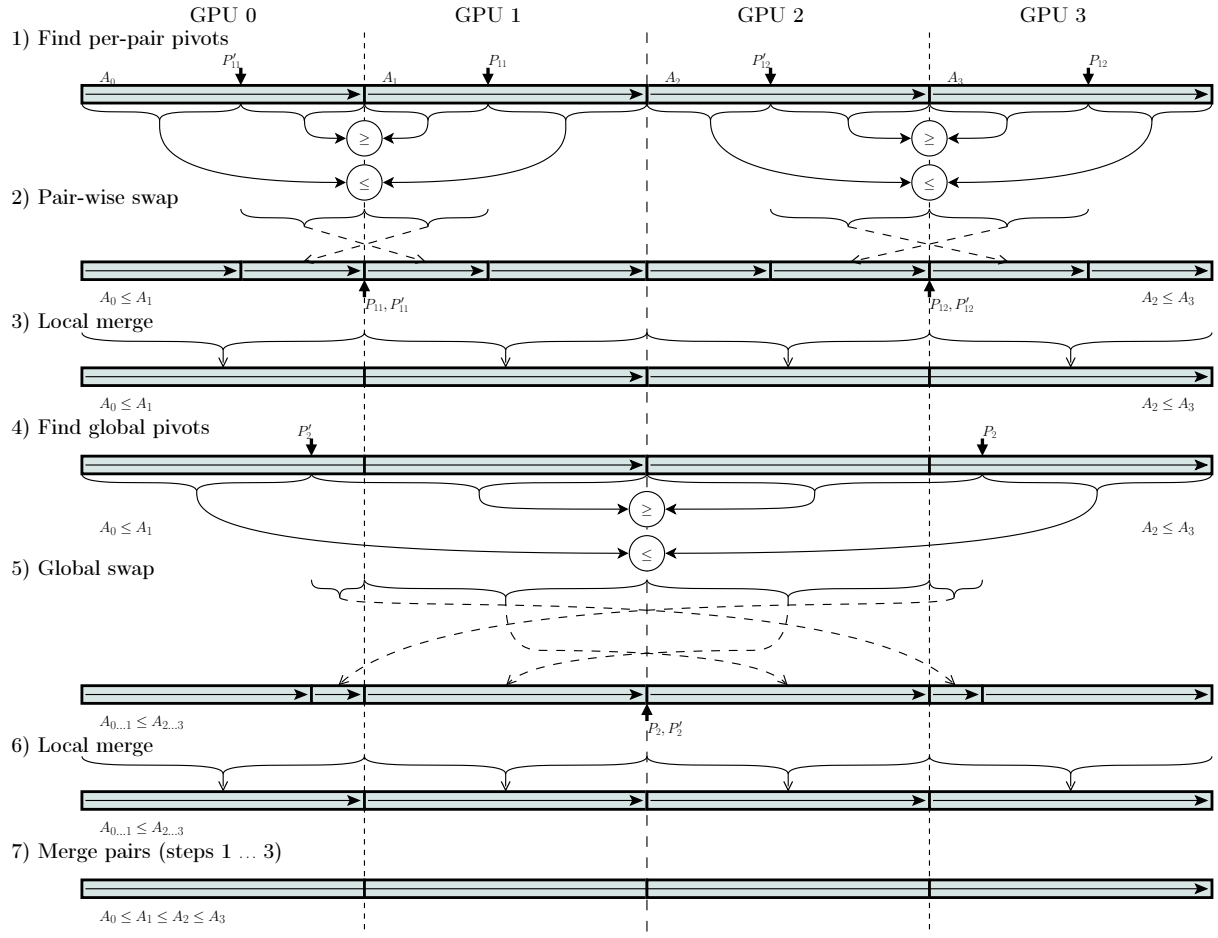


Figure 3: Execution of the algorithm when sorting on 4 GPUs.

to be found for each pair of GPUs (step 1 in Figure 3). Then, the "inner" sub-arrays identified by the pivots are swapped between pairs of GPUs (step 2 in Figure 3), and a merge is performed locally by each GPU (step 3 in Figure 3). As a result, each pair of GPUs contain one sorted array ($A_0 \leq A_1$ and $A_2 \leq A_3$) due to the pivot properties stated in Observation 1.

Subsequent stages need to find pivots for arrays that span two or more GPUs. In the case of four GPUs (steps 4 through 7 in Figure 3), a pivot has to be found anywhere in the space comprised by two sorted arrays distributed in two GPUs (as well as the mirrored pivot in the other two GPUs). After the pivots are found (step 4 in Figure 3), the "inner" records of the sorted arrays are swapped (step 5 in Figure 3). Because pivots can fall in any of the GPUs, two or four GPUs (depending on where the pivots fall) will have to swap some of the records with their peers. After swapping records, all the elements in A_0 and A_1 are smaller than or equal to any element in A_2 and A_3. The array relationships $A_0 \leq A_1$ and $A_2 \leq A_3$ accomplished in step 3 in Figure 3 no longer hold true, but now it holds that $A_{0\ldots1} \leq A_{2\ldots3}$. Steps 6 and 7 in Figure 3 ensure that we obtain the final globally sorted array where $A_0 \leq A_1 \leq A_2 \leq A_3$. This same procedure can be repeated as many times as necessary to perform a merge sort using an arbitrary number of GPUs.

3.4 Pivot selection

To efficiently find the pivot, we utilize a binary search like pattern where candidate pivots are picked from one array and compared with the records at the "mirrored" index in the other array (see Algorithm 1).

Algorithm 1 Pivot selection

```
function PIVOT_SELECTION(array a, array b)
    pivot ← len(a)/2
    stride ← pivot/2
    while stride > 0 do
        if a[len(a) − pivot − 1] < b[pivot] then
            if a[len(a) − pivot − 2] < b[pivot + 1]  &
                    a[len(a) − pivot] > b[pivot − 1] then
                return pivot
            else
                pivot ← pivot − stride
            end if
        else
            pivot ← pivot + stride
        end if
        stride ← stride/2
    end while
    return pivot
end function
```

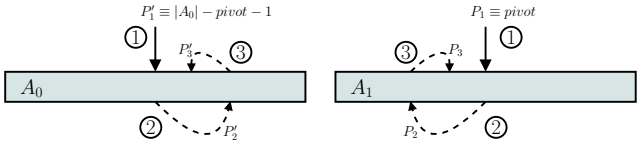


Figure 4: Pivot selection that results in the correct pivot after three steps.

First, the middle elements of each sub-array are selected as candidate pivots. If these values fulfill the stop condition that the keys smaller than the pivot in one array are smaller than their mirrored keys in the other array, and the keys bigger than the pivot in one array are bigger than their mirrored keys in the other array, the algorithm stops. If the pivots lie further apart or closer than the current candidate pivots, a new pair of pivots is selected. New pivots are picked by adding or subtracting (depending on the comparison of candidate pivots outcome) the next candidate distance to the indexes of the current candidate pivots. Like in a binary search, the distance is getting halved in each step, thus leading to a logarithmic complexity. The pattern is depicted in Figure 4 for the case where pivots are found in three steps.

4. IMPLEMENTATION DETAILS

Even though our algorithm is designed for multi-GPU systems from the ground up, the implementation still had to overcome some of the practical limitations of current systems. Here, we present the details of the implementation along with the novel technique for communication of GPUs on different PCIe buses that allowed us to implement the algorithm efficiently.

4.1 Inter-GPU Communication: Pivot Selection

As described in Section 3, the pivots are found using an algorithm similar to binary search. This operation can be implemented in parallel by splitting the input arrays in smaller chunks, performing independent local pivot selections and using a reduction to obtain the final pivot position. However, a parallel implementation of the pivot selection would provide little benefit, if any, since this is a very fast operation whose contribution to the total execution time is marginal. Because of this, we implement a single-threaded version of the pivot selection algorithm to keep our source code as simple as possible, without paying the performance penalty. When P2P access between two GPUs is **available**, we implement pivot selection using a single GPU. The code uses regular load instructions to read the values from both the local and the remote GPU memories, which are compared to determine if any of the read values is the pivot. This serial implementation of pivot selection requires, at most, $log_2(n)$ steps (i.e., 30 steps for 1 Giga records), and each step performs only three P2P memory accesses. The little contribution to the total execution time (around 0.01% for the case of two GPUs) makes the serial pivot implementation a competitive approach. When P2P access between two GPUs is **not available**, we emulate it using host-mapped memory, as illustrated in Figure 5. We allocate four host memory variables: two synchronization variables, a variable to store the index of candidate pivots and a small buffer to store the keys. Because all these variables are allocated

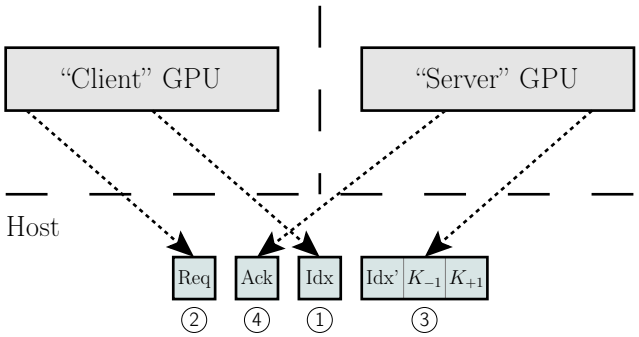


Figure 5: Emulated P2P access through host-mapped memory.

in host-mapped memory, any GPU or CPU in the system can read and write to these variables. In this process, one GPU is designated to be the client while the other GPU is the server. The client asks the server for the candidate key by writing the desired index to the shared variable (step 1 in Figure 5), and setting the request variable (step 2). After this, the client starts busy waiting (polling the synchronization variables) until the request is served. The server GPU busy waits until a request is received. When this happens, it reads the index variable, writes the candidate pivot and the two surrounding keys (for optimization purposes) to the buffer (step 3), and signals the client that the request is served by setting the acknowledge variable (step 4). This process is repeated until the client finds the pivot, after which the server is signaled to stop. A combination of native and emulated P2P accesses is used when the keys are spread across more than two GPUs in which some can use the P2P access and some cannot. This operation is less efficient than the case where P2P accesses between two GPUs are possible, but it still contributes very little (around 0.07% for the case of four GPUs) to the overall execution time.

4.2 Data Transfers: Merge Phase

Once the pivot is found, the CPU needs to read it so that it can orchestrate the swapping of data between GPUs. Instead of issuing a memory transfer from device to host memory, just to transfer the pivot index, we optimize this transfer by using host mapped memory so that the CPU can access the pivot as soon as the pivot selection kernel finishes executing. After that, portions of the records between the two sorted arrays are exchanged. We exploit that merge sort is not an in-place algorithm, and thus each GPU requires two equally sized internal buffers, one for the input data and the other one for the output data of each merge stage. Our data swapping routine uses both buffers: records coming from another GPU are written to the idle buffer, which is also filled with the remaining records from the other local buffer. Since the data transfer inside the device memory is fast, compared to transfer between two GPUs, this technique leads to faster overall execution than in-place swap, because it avoids the overhead of synchronization. This data exchange phase might easily become an important bottleneck if not implemented in an efficient way. Thus it is crucial that data transfers exploit the full-duplex capabilities of the interconnects (i.e., PCIe bus and QPI in our system). This is achieved by overlapping data transfers during the data

swapping phase, i.e., each pair of GPUs are simultaneously sending and receiving data. When available, asynchronous P2P memory transfer commands are issued over two separate CUDA streams. This implementation achieves the two aforementioned goals: maximization of the PCIe bandwidth, and overlapping of data transfers in both directions.

Since the P2P memory transfers are not possible across GPUs connected to separate PCIe buses, a temporary buffer in the host memory is used in this case. The data is first transfered from the source GPU, and then send to the destination GPU thus enabling the emulation of P2P memory transfers and offering a common API to programmers. During the implementation we have detected that the CUDA provided implementation of emulated P2P memory transfers is not NUMA aware (see Section 2), which produces an underutilization of the available interconnect bandwidth. We implement our own data transfer routine that spawns one host thread per GPU in the system, and pins each host thread to the CPU connected to the same PCIe bus as the GPU. This ensures that the operating system schedules threads for execution only on the set of optimal CPUs, and that all the host memory allocations of the thread are performed in the memory local to the thread's CPU. We also double buffer each data transfer to mitigate the performance penalty of performing two DMA transactions, i.e., from the source GPU memory to host memory, and from host memory to the destination GPU. We use pinned memory for the intermediate host memory buffers to avoid intermediate memory copies between user-level threads and the operating system kernel. The combination of all these implementation techniques, allows our data transfer code to achieve a higher data transfer bandwidth (around 60%) than the default implementation provided by CUDA.

5. EXPERIMENTAL EVALUATION

To show the effectiveness of our proposed algorithm and communication techniques, we analyze the performance and scalability of our implementation, compare it with a multi-core CPU implementation of sorting, and analyze the effects of key size on the performance.

Our test system consists of four NVIDIA Tesla C2070 GPUs clocked at 1.15 GHz, with 6GB of memory each, two Intel Xeon E5620 CPUs with four cores (each core is two-way SMT) clocked at 2.4 GHz, and 24GB of RAM memory on the host. Two PCIe 2.0 x16 buses (two GPUs per bus) are present in the system and the CPUs are interconnected with Intel Quickpath Interconnect (like in Figure 1). On the software side, we are running Linux with the 3.2.0 kernel, CUDA runtime version 4.1, NVCC version 4.1 and GCC version 4.5.3.

All measurements are repeated 32 times and the average value is shown together with the confidence interval of $\pm$ standard deviation (envelopes in Figure 6). The number of records that can be sorted is limited only by the available GPU memory. To allow the single GPU sorting routine to efficiently use the available memory bandwidth by coalescing memory accesses, we follow the approach of splitting the records into the pairs of keys and values proposed in [10] and adopted by recent publications on GPU sorting. We have tested our system with both integer and floating point keys, and observed no difference in performance for keys of the same size. Because of that, we report the results for floating point keys only (single and double precision). In all experiments, values are 32 bits.

5.1 Sorting Performance

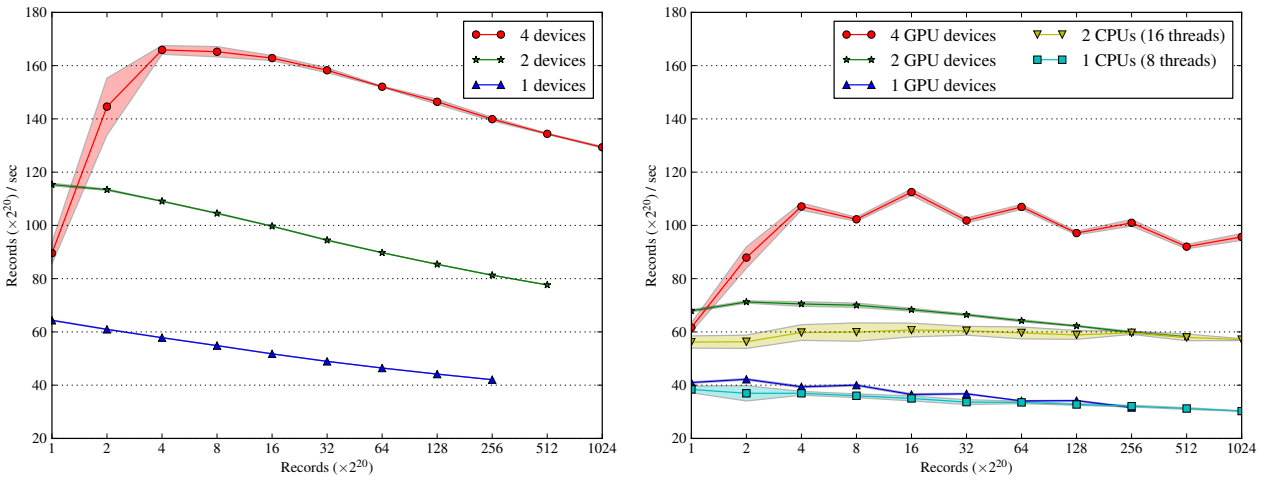
Figure 6 shows the sorting rates when using multiple GPUs and CPUs. Sorting rates for one, two and four GPUs without initial data transfer from the host and final data transfer from the GPUs are shown in Figure 6(a), while sorting rates for one and two CPUs and one, two and four GPUs with these transfers accounted for are shown in Figure 6(b). Keys are 32-bit floating point numbers (single precision), generated randomly with a uniform distribution. The total number of sorted records ($\times 2^{20}$), regardless of the number of processing units (GPUs or CPUs) used, is shown on the x axis, while the sorting rate ($\times 2^{20} records/s$) is on the y axis.

As already mentioned in Section 1, applications usually sort data produced by previous stages and consumed by next stages. This scenario is covered in Figure 6(a). Configurations with two and four GPUs are able to sort two and four times more records (512 mega records and 1024 mega records) than the single GPU is able to (256 mega) due to the doubled and quadrupled amount of available memory. The sorting rate for a single GPU peaks at 1 mega records, followed by a slight decreasing trend due to the $n \log n$ nature of the merge sort. The sorting rates on two and four GPUs closely follow that trend and reach the peak performance when sorting around 1 mega records per GPU. Sorting on four GPUs does not reach the peak performance for datasets smaller than 4 mega records because the single-GPU sorting does not reach the peak until that point and because of the synchronization overhead imposed by the CPU threads that are controlling the execution on GPUs. There are more synchronization points and more threads are participating in the case of sorting on four GPUs than in the case of two. This also results in a higher variability of sorting rate in the configuration with four GPUs (thicker envelope in Figure 6(a) and Figure 6(b)).

When data transfers to and from the host are accounted for (shown in Figure 6(b)) sorting rates are clearly lower than the ones without data transfers (Figure 6(a)) due to overhead that these data transfers introduce. It is also worth noting that the sorting rates for one and, especially four GPUs in Figure 6(b) vary significantly from one input size to the other, which is explained later in the Section 5.4

5.2 Comparison with CPU Sorting

We also compare the sorting rates of a multi-core sorting implementation with our multi-GPU implementation for the same sets of records in Figure 6(b). In this case, we included the data transfers to the GPU memory and back to the CPU memory in the measurements. For the CPU sorting algorithm, we use the parallel version of the sorting routine from GCC libstd++ [28] which is also a two-phase merge sort, like ours. Each thread first sorts independent chunks of data using a fast single threaded sorting algorithm (introspective sort [29]) and then merges the sorted chunks. We measured the sorting rate of the CPU algorithm when executing on one CPU (four cores, eight threads) and both CPUs (eight cores, sixteen threads) in our system by setting the affinity of the operating system process to only one of the CPUs or both. The results show that for the compared algorithms in our system, using the same number of GPUs



(a) No data transfers included

(b) With data transfers included

Figure 6: Sorting rates of one, two and four GPUs and one and two CPUs.

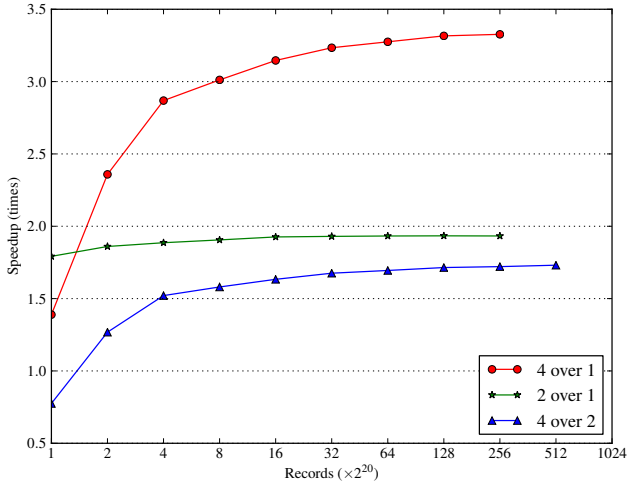


Figure 7: Strong scalability of the proposed solution using 2 and 4 GPUs.

as CPUs results in similar sorting rates when the data is in the CPU memory. GPU sorting is slightly faster when comparing one CPU and one GPU for number of records smaller than 64 mega records (up to 10% for 2 mega records) and when comparing two CPUs with two GPUs for number of records smaller than 256 mega records (up to 20% for 2 mega records). When using all four available GPUs or both available CPUs, sorting on the GPUs is clearly faster for 2 or more mega records (up to 80% for 16 mega records). Hardware configurations similar to our test system (more GPUs than CPUs) are becoming more common in many machines designed to exploit GPU's high performance per watt. In this case, applications can use all the GPUs in the system to achieve a higher sorting rate than with the CPU version, which makes this work even more relevant.

5.3 Scalability Analysis

Figure 7 shows the speedup obtained when sorting on two and four GPUs, compared to sorting on one GPU and the speedup obtained when sorting on four GPUs compared to sorting on two for the sorting rates from Figure 6(a). The total number of records to be sorted (x axis) is distributed among all the GPUs that participate in the sorting, thus the strong scalability of the algorithm is measured. As can be seen from the figure, speedup when using two GPUs instead of one is more than 1.7x and approaches the peak of 1.9x as the number of records to be sorted grows. The speedup when using four GPUs over two is visible from 2 mega records, and reaches 1.7x for 512 mega records. This results in the speedup of more than 3x for 8 or more millions of records and the maximum obtained speedup of 3.3x (for 256 mega records) when using four GPUs compared to the base single-GPU case. The performance improvement when using four GPUs instead of two is smaller than the performance improvement obtained when using two GPUs instead of one due to two main factors. First, when using two GPUs, all the communication and data transfers in our system go through the PCIe bus which is utilized efficiently. When using four GPUs, the communication and transfers are done through the PCIe and the processor interconnect (QPI in our case) which is not being fully utilized, as explained in Section 2. Second, because the data is merged across four GPUs (four physical domains), the problem is first reduced, via one merge stage, to two smaller and independent (two GPUs) problems and then finally solved, as explained in Section 3. Other than speedup, an equally important effect of increasing the number of GPUs is that the total amount of data that can be sorted is also being increased, because the cumulative device memory has increased.

5.4 Execution Breakdown and Sensitivity to Key Size

Figure 8 shows the ratio of time spent in data transfers with the host (labeled as "Host to Device" and "Device to Host") and all of the sorting phases (labeled as "Sorting") when using four GPUs. Records have 32 bit single precision (SP) or 64 bit double precision (DP) floating point keys and 32 bit integer values. Data transfers take from 17% to 32%

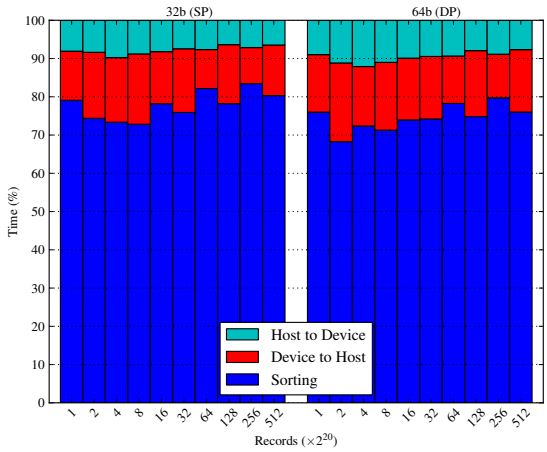


Figure 8: Time spent in data transfers and sorting on four GPUs for 32 bit (Single Precision) and 64 bit (Double Precision) keys and different total amount of records. In all configurations, values are 32 bit integers.

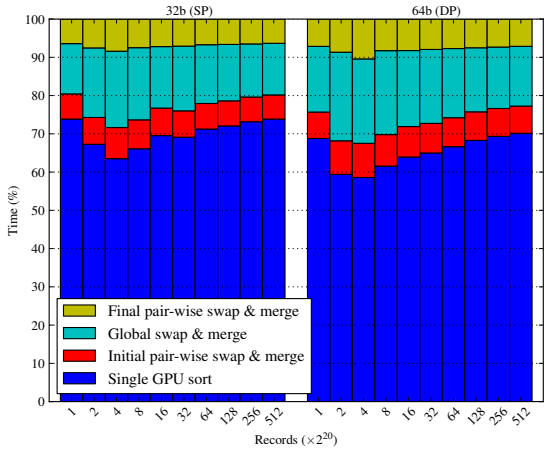


Figure 9: Execution breakdown of the sorting on four GPUs for 32 bit (Single Precision) and 64 bit (Double Precision) keys and total numbers of records. In all configurations, values are 32 bit integers.

of the execution time and, as can be seen, the reasons for the varying sorting rate in Figure 6(b) are the slow memory transfers from device to host for some data sizes. We have also observed similar behavior in other multi-GPU applications we examined and we believe that this is a driver or runtime issue and thus should be fixed in future releases of the NVIDIA CUDA toolkit. This behavior can not be seen in Figure 6(a) because it shows the sorting rate without transfers from and to host.

Execution of the actual sorting is further split and shown in Figure 9. The execution time is dominated by sorting partitions on each GPU using the single-GPU sorting algorithm (from 60% to 70% on average). Pairwise swap and pairwise merge in both initial and final steps take roughly the same amount of time, and are collapsed here for clarity into "Initial pair-wise swap & merge" and "Final pair-wise swap & merge". Global swap and merge stages are also collapsed into "Global pair-wise swap & merge", which is dominated by the global swap (roughly 80% of the stage). Pair-wise pivot selection with P2P access (0.01%) and four-GPU pivot selection with emulated P2P access (0.07%) take an insignificant amount of time (not shown separately in the graph). Because the sorting rate for one GPU reaches the peak performance for 1 mega records, the percentage of time spent in single-GPU sorting decreases until 4 mega records (1 mega records per GPU) after which it slowly increases. If the inter-GPU communication over the processor interconnect was fully utilized, transfers would be faster (as shown in Figure 2), resulting in an up to 30% shorter "Global swap" stage. The number of necessary communication steps would remain the same. The breakdown of the execution time for two GPUs is not shown separately, but can be extracted from Figure 9, taking into account only the single-GPU sort and initial pair-wise swap and merge stages. In that case, the bulk of the time is spent in single-GPU sorting, while swap and merge stage take from 7% to 13%. Note that the number of records is also halved (the x axis shows the number of records spread across all four GPUs).

Changing keys from 32 bit to 64 bit floating point results in 9% to 11% lower sorting rate for the single-GPU sorting stage. Due to the 50% increase in the size of the record (key size doubles, values stay the same), data transfers take 50% more time. This results in an overall sorting rate of 79% to 85% (with an average of 83%) of the sorting rate obtained with the 32 bit keys.

6. RELATED WORK

Researchers have been working on sorting mechanisms for parallel machines, both theoretical and practical, for decades. Batcher was the first one to describe an elegant parallel sorting mechanism in the form of bitonic sorting network [30]. Later, as the interest in parallel algorithms started growing, sorting got a lot of attention, being a fundamental, yet hard, problem to solve efficiently. Most of these algorithms were proposed for different variations of the theoretical Parallel Random Accesses Machine model, some of which include [31, 27].

Several research efforts have also explored how to port parallel sorting algorithms to specific systems. Often time, this research came to different conclusions, because specific features of a given machine will determine which algorithm is the optimal. For example, Blelloch et al. [32] evaluated sample sort, radix sort and Batcher's bitonic sort on a massively parallel machine, Thinking Machine CM-2. They concluded that radix sort is the fastest, while sample sort is the fastest comparison based sort on their target machine. Solomonik et al. [33] compared histogram sort, merge sort, and radix sort on two modern, homogeneous supercomputers. This work concluded that an improved histogram sort, proposed by the authors, was the fastest algorithm on both machines. Satish et al. [10] compared their merge sort with radix sort on one of NVIDIA's previous architectures (GT200), concluding that radix sort was faster than merge sort. Satish et al. [34] compared merge sort and radix sort on CPUs and GPUs and showed that for the future large SIMD widths, merge sort is the more promising approach.

Adoption of GPUs for general purpose computing resulted

in a number of papers on sorting on the GPUs. First approaches of comparison based sorting on GPUs were mainly implementations of various sorting networks. Cederman et al. [13] proposed an efficient quick sort algorithm on GPUs. Satish et al. [10] proposed implementations of radix sort and merge sort algorithms. Their implementation of merge sort based on [27] was a significant improvement over previous approaches, while their radix sort implementation was significantly faster, and still is one of the fastest sorting algorithms on GPUs. Xiaochun et al. [11] proposed an optimization technique in which they use buckets of warp size instead of thread block size. Because this technique omits explicit synchronization in one thread block (warps are implicitly synchronized) [16] they were able to obtain speedup over a merge sorting network and the merge sort algorithm proposed in [10] when applying this technique. Leischner et al. [12] proposed an implementation of sample sort which was the fastest implementation of a comparison based sort for the NVIDIA GT200, architecture. Unlike previous approaches that target only keys of fixed length, Davidson et al. [35] proposed a merge sort based algorithm for keys with variable length. All of these algorithms were proposed for system with a single GPU and all of them can be used as a first step of our algorithm (to sort data in each GPU). Green et al. [36] proposed a novel single-GPU merging algorithm which can also be used in our algorithm for the local, single-GPU, merges to further improve the performance.

To the best of our knowledge, there is no GPU sorting algorithm designed for multi-GPU systems. Sintron et al. [9] proposed a two phase algorithm for GPUs which they also adopted for use with two GPUs. The first phase consists of the distribution of elements to their destination buckets, followed by sorting the buckets independently with merge sort. The drawback of this method is that the process of calculating the buckets is performed by the CPU and results in a series of communications between GPU and CPU to refine the initial partition selection. Also, this scheme uses an additional array to store the position of each element before moving them to the buckets, utilizing the available memory inefficiently. The algorithm is proposed for the first generation of CUDA capable GPUs and evaluates it analytically, disregarding the communication costs and the limitations of the real machine. There were also efforts to propose external sorting algorithms in which the data size is larger than the GPU memory capacity, but it fits in the backing memory (system memory in this case). Peters et al. [15] proposed sorting chunks of data independently by the GPU, bringing them one by one back to the host memory, partitioning them using the CPU, and then returning the partitions to the GPU to merge them. Ye et al. [14] adopted the deterministic sample-based parallel sorting [37] in their external sorting algorithm. They also proposed an innovative strategy to quickly select good samples for the algorithm they use for sorting inside the GPU. Both of these external sorting algorithms can be naturally extended to use multiple GPUs by sorting independent chunks on multiple GPUs in parallel, but both of them are designed to sort large amount of data that does not fit into GPU's memory, and this is when they perform the best. Unlike our efficient pivot selection algorithm, none of the solutions that can use more than one GPU can split the input data to be sorted perfectly among GPUs, thus none of them can fully utilize the available memory in the GPU. Furthermore, all three solutions utilize execution on the CPU to some extent.

7. CONCLUSIONS

The importance of porting fundamental algorithms, like sorting, on emerging platforms is two-fold. On one hand it allows other algorithms that depend on them to be ported to these emerging platforms. On the other hand it also evaluates the features and flaws of these platforms. In this paper we have presented a high performance merge sort algorithm for multi-GPU systems which suits the need for efficient sorting operations in current and future systems comprised of several CPUs and GPUs.

A key part of the work is a pivot selection algorithm that enables sorting data structures spread across the memory of all the GPUs in the system by ensuring that at any stage of the algorithm data is always evenly distributed. Another key point of this work is the insight on how to efficiently implement the proposed algorithm using CUDA for NVIDIA GPUs. In our implementation we have introduced a novel inter-GPU communication scheme for GPUs connected to separated PCIe buses. This technique is not specific to our sorting implementation and can be applied to other algorithms that require inter-GPU communication over interconnection networks that do not directly support it.

Experimental evaluation of our algorithm shows that it scales to 1.9x when moving from a single-GPU implementation to a system with two GPUs and 3.3x when moving to a system with four GPUs while allowing the dataset to double and quadruple in size. When the data is in the GPU, it outperforms analogous sorting algorithms running on multi-core CPUs when comparing the same number of CPUs against GPUs in our test system. It also shows performance slightly better when the data resides in the CPU memory. This means that our algorithm is not only applicable to cases where the data to be sorted is already in the GPU memory (enabling GPU-based algorithms to use an intermediate sorting step in the GPU), but also to speed up both CPU and GPU applications in the "GPU-heavy" systems (more or higher class GPUs than CPUs) that rely on the sorting operation.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their time and valuable comments. This work is supported by European Commission through TERAFLUX (FP7-249013) and Mont-Blanc (FP7-288777) projects, NVIDIA through the CUDA Center of Excellence program, Spanish Ministry of Science and Technology through Computacion de Atlas Presteciones V and VI (TIN2007-60625 and TIN2012-34557) and University of Illinois at Urbana Champaign.

9. REFERENCES

[1] P. Jetley, F. Gioachin, C. Mendes, L. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[2] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 18:1–18:11.

[3] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[4] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, Sep. 2006.

[5] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using gpus," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, ser. UCHPC-MAW '09, 2009, pp. 1–6.

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, 2008, pp. 260–269.

[7] J. Stuart and J. Owens, "Multi-gpu mapreduce on gpu clusters," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 1068 –1079.

[8] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, may 2008.

[9] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381 – 1388, 2008.

[10] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, may 2009, pp. 1 –10.

[11] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, "High performance comparison-based sorting algorithm on many-core gpus," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –10.

[12] N. Leischner, V. Osipov, and P. Sanders, "Gpu sample sort," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –10.

[13] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 4:1.4–4:1.24, Jan. 2010.

[14] Y. Ye, Z. Du, and D. A. Bader, "Gpumemsort: A high performance graphic co-processors sorting algorithm for large scale in-memory data," 2010.

[15] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Parallel external sorting for cuda-enabled gpus with load balancing and low transfer overhead," *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, vol. 0, pp. 1–8, 2010.

[16] NVIDIA, *CUDA C Programming Guide*, 2012.

[17] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, January 1962.

[18] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998, vol. 3 Sorting and Searching, noted by the author in p. 158.

[19] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10, 2010, pp. 94–103.

[20] "Amd fusion whitepaper." [Online]. Available: http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf

[21] "Intel quickpath interconnect," 2012. [Online]. Available: http://www.intel.com/technology/quickpath

[22] "Hypertransport interconnect," 2012. [Online]. Available: http://www.hypertransport.org

[23] "Heterogeneous system architecture," 2012. [Online]. Available: www.hsafoundation.com

[24] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 347–358.

[25] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. mei Hwu, "Gpu clusters for high-performance computing," in *Workshop on Parallel Programming on Accelerator Clusters. IEEE CLUSTER '09*, 31 2009-sept. 4 2009, pp. 1 –8.

[26] NVIDIA, "Thrust parallel algorithms library," 2012. [Online]. Available: http://thrust.github.com

[27] T. Hagerup and C. Rüb, "Optimal merging and sorting on the erew pram," *Information Processing Letters*, vol. 33, no. 4, pp. 181 – 185, 1989.

[28] J. Singler, P. Sanders, and F. Putze, "Mcstl: The multi-core standard template library," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Springer Berlin / Heidelberg, 2007, vol. 4641, pp. 682–694.

[29] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[30] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring), 1968, pp. 307–314.

[31] R. Cole, "Parallel merge sort," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, oct. 1986, pp. 511 –516.

[32] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "An experimental analysis of parallel sorting algorithms," *Theory of Computing Systems*, vol. 31, pp. 135–167, 1998.

[33] E. Solomonik and L. Kale, "Highly scalable parallel sorting," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –12.

[34] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10, 2010, pp. 351–362.

[35] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[36] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12, 2012, pp. 331–340.

[37] D. R. Helman, J. JáJá, and D. A. Bader, "A new deterministic parallel sorting algorithm with an experimental evaluation," *J. Exp. Algorithmics*, vol. 3, Sep. 1998.