GPU Merge Path - A GPU Merging Algorithm

Oded Green [*]
Georgia Institute of Technology

Robert McColl
Georgia Institute of Technology

David A. Bader
Georgia Institute of Technology

ABSTRACT

Graphics Processing Units (GPUs) have become ideal candidates for the development of fine-grain parallel algorithms as the number of processing elements per GPU increases. In addition to the increase in cores per system, new memory hierarchies and increased bandwidth have been developed that allow for significant performance improvement when computation is performed using certain types of memory access patterns.

Merging two sorted arrays is a useful primitive and is a basic building block for numerous applications such as joining database queries, merging adjacency lists in graphs, and sorting. An efficient parallel algorithm for merging partitions the sorted input arrays into sets of non-overlapping sub-arrays that they can be independently merged on multiple cores. For optimal performance, the partitioning should be done in parallel and should divide the input arrays such that each core receives an equal size of data to merge.

In this paper, we present an algorithm that meets the partitions the workload equally amongst the GPUs multi-processors. Following this, we show how each multi-core processor performs a parallel merge. All stages in this algorithm are parallel. We also show how this algorithm utilizes the GPU memory hierarchy in the different stages. We show an average of 20X and 50X speedup over a sequential merge on the x86 platform for integer and floating point, respectively. Our algorithm is 10X faster than the merging algorithm in the CUDA Thrust library.

Keywords

Parallel algorithms, Parallel systems, Graphics processors, Measurement of multiple-processor systems

1. INTRODUCTION

The merging of two sorted arrays into a single sorted array is straightforward in sequential computing, but presents challenges when performed in parallel. Given that merging is a common building block for larger applications such as database querying and management, graph contraction, sorting, etc., improving performance through parallel computing approaches can provide benefit across disciplines. Given two sorted arrays A, B of length $|A|, |B|$ respectively, the output of the merge is a third array C such that C contains the union of elements of A and B, is sorted, and $|C| = |A| + |B|$. The computational time of this algorithm on a single core is $O(|C|)$ [2].

The increase in the number of cores in modern computing systems present numerous challenges that need to be met for an algorithm to achieve good performance. These challenges include equal partitioning for effective load balancing, reducing the need for synchronization mechanisms and minimizing redundant operations brought forth by the parallelization. We will present an algorithm that meets all these challenges and more as it is aimed at GPU systems.

The remainder of the paper is organized as follows. In this section we present a brief introduction to GPU systems, merging, and sorting. Section 2 introduces the new GPU merging algorithm, GPU Merge Path, and presents the different granularities of parallelism. In section 3, we show empirical results of the new algorithm on two different GPU architectures. Section 4 offers concluding remarks.

1.1 Introduction on GPU

Graphics Processing Units (GPUs) are becoming a popular platform for parallel computation in recent years following the introduction of programmable graphics architectures like CUDA [6] that allow for easy utilization of the cards for purposes other than graphics rendering. For the sake brevity, we will not present the entire CUDA architecture here but rather a short introduction. For more reading on GPU sorting, we refer the reader to [8], [4], and [1].

The original purpose of the GPU was to accelerate graphics applications which are highly parallel and computationally intensive, thus, having a large number of simple cores can allow the GPU to achieve high throughput. These simple cores also known as SP cores and they are arranged into groups of 8/16/32 (depending on the CUDA compute capability) cores known as SM multi-threaded multiprocessors. The exact number of SMs on the card is system. Each SM has a single control unit responsible for fetching and decoding instructions. All the SPs for a single SM will execute the same instruction at a given time but on different data or perform no operation in that cycle. Thus, the parallel granularity is limited by the number of physical SPs. The SMs are responsible for scheduling the threads to the

[*]Corresponding author : ogreen@gatech.edu

26th International Conference on Supercomputing '12 San Servolo Island, Venice, Italy

SPs. The threads are executed in groups called warps. The current size of a warp is 32 threads. Each SM has a shared-memory/private cache. Older generation GPU systems have 8KB shared-memory, whereas the new generation has 64KB of shared-memory which can be used in two separate modes.

Programming for the GPU requires defining functions called kernels. CUDA also requires dividing the total number of threads into smaller units called thread blocks. As the thread blocks are sent to the SMs for execution, it is best to create threads blocks that are multiples of the warp size as this promises to utilize the SPs and in mosts cases increases performance. As each thread block is executed by a single SM, the thread blocks can use shared data with the shared-memory of the SM. A thread block is considered complete when all the threads in that specific block have completed. Only when all the threads blocks have completed execution is the kernel considered complete.

1.2 Parallel Sorting

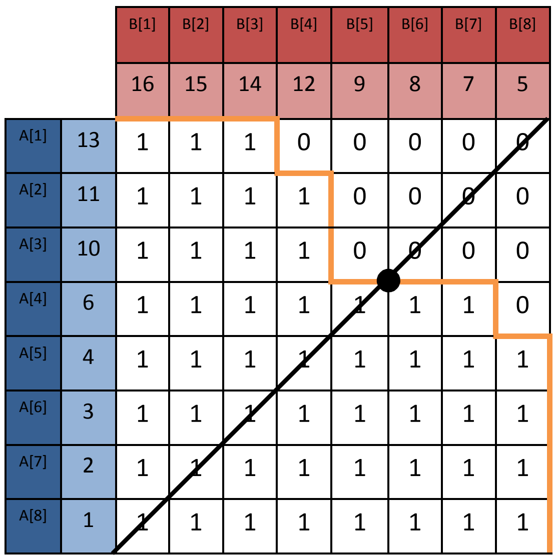
While our focus in this paper is not on sorting, but rather on merging, we present a brief description of prior work in the area of sorting: sequential, PRAM sorting, and GPU sorting [8, 10]. In further sections there will be a more thorough background on parallel merging algorithms.

Sorting is a key building block of many algorithms. It has received a large amount of attention in both sequential algorithms (bubble, quick, merge, radix) [2] and their respective parallel versions. Prior to GPU algorithms, several merging and sorting algorithms for PRAM were presented in [3, 7, 9] . Following the GPGPU trend, several algorithms have been suggested that implement sorting using a GPU for increased performance. Some of the new algorithms are based on a single sorting method such as the radix sort in [8]. In the same paper, the authors suggest using a parallel merge sort algorithm that is based on a division of the input array into sub arrays of equal size followed by a sequential merge sort of each sub array. Finally, there is a merge stage where all the arrays are merged together in a pair-wise merge tree of depth $log(p)$. A good parallelization of the merge stage is crucial for good performance. Other algorithms use hybrid approaches such as [10], which uses bucket sort followed by a merge sort. One consideration for this is that each of the sorts for a particular stage in the algorithm is highly parallel, which allows for high system utilization. Additionally, using the bucket sort allows for good load balancing in the merge sort stage; however, the bucket sort does not guarantee an equal work load for each of the available processors and – in their specific implementation – requires atomic instructions.

1.3 Serial Merge

While merging is a well-known algorithm, it is necessary to present it due to a reduction that is presented further in this section. Given arrays A, B, C as defined earlier, a simplistic view of a decreasing-order merge is to start with the indices of the first elements of the input arrays, compare the elements, place the larger into the output array, increase the corresponding index to the next element in the array, and repeat this comparison and selection process until one of the two indices is at the end of its input array, then copy the remaining elements from the other input array into the end of the output array [2].

In [7], the authors suggests treating the sequential merge as though it were a path that moves from the top-left corner of an $|A| \times |B|$ grid to the bottom-right corner of the grid. The path can move only to the right and downwards. The reasoning behind this is that the two input arrays are sorted. This ensures that if $A[i] > B[j]$, then $A[i] > B[j']$ for all $j' > j$. In Fig. 1, $A[1] = 13$ is greater than $B[4] = 12$, thus, it is greater than all $B[j]$ for $j > 4$. We mark these elements with a '0' in the matrix. In practice the translates to marking all the elements in the first row and to the right of $B[4]$(and including) with a '0'. Consequently if $A[i] > B[j]$, then $A[i'] > B[j]$ for all $i' < i$. Fig. 1, $A[3] = 10$ is greater than $B[5] = 9$, as are $A[1]$ and $A[2]$. All elements to the left of $B[4] =$ are marked with a '1'. The same inequalities can be written for B with the minor difference that the '0' is replaced with '1'.



		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		16	15	14	12	9	8	7	5
A[1]	13	1	1	1	0	0	0	0	0
A[2]	11	1	1	1	1	0	0	0	0
A[3]	10	1	1	1	1	0	0	0	0
A[4]	6	1	1	1	1	1	1	1	0
A[5]	4	1	1	1	1	1	1	1	1
A[6]	3	1	1	1	1	1	1	1	1
A[7]	2	1	1	1	1	1	1	1	1
A[8]	1	1	1	1	1	1	1	1	1

Figure 1: Illustration of the MergePath concept for a non-increasing merge. The first column (in blue) is the sorted array A and the first row (in red) is the sorted array B. The gold path (a.k.a. Merge Path) represents comparison decisions that are made to form the merged output array. The black cross diagonal finds the median of the output array, which is its intersection with the gold path and allows for a 2-way parallelization.

An interesting point is that the path that follows exactly along the boundary between the '0's and '1's represents the selections from the input arrays that form the merge. This is depicted in Fig. 1 with the orange, the heavy stair-step, line. It should be noted here that the matrix of '0's and '1's is simply a convenient visual representation and is not actually created as a part of the algorithm (i.e. the matrix is not maintained in memory and the comparisons to compute this matrix are not performed).

In this way, performing a merge can be thought of as the process of discovering this path. When $A[i] \geq B[j]$ move down by one position and copy $A[i]$ into C, else when $A[i] < B[j]$ move to the right and copy $B[j]$ into C. The opposite is also true, if the path has gone down at some point it means that $A[i] \geq B[j]$ and that any merging algorithm would select $A[i]$ before $B[j]$. From a pseudo-code point of view, moving to the right can be considered taking the branch/condition when $A[i] \leq B[j]$ and the moving down can be thought of taking the else of that branch. Consequently, given this path the order in which the merge to be completed is totally known. Thus, all the elements can be placed in the output array in parallel based on their position(sum of row and column indices of an element) in the path.

1.4 GPU Challanges

To achieve maximal speedup on the GPU platform, it is necessary to implement a platform dependent(and in some cases, card dependent) algorithm. These implementation are architecture dependent and in many cases require a deep understanding of the memory system, interconnect, and the execution system. Ignoring the architecture will limit the achievable performance. For example, a well known performance hinderer is bad memory access (read/write) patterns to the global shared memory.

For good performance, all the SPs on a single MP should read/write sequentially. If the data is not sequential (different memory lines in the DRAM), this could lead to multiple memory requests which causes all SPs to wait for all memory requests to complete. One way to achieve the efficient memory use is to do a sequential data read into the shared-memory incurring one memory request to the global memory and follow with 'random'(non-sequentual) memory accesses to the shared-memory.

An additional challenge is to find a way to divide the workload evenly among the SMs and further partition the workload amongst the SPs. Improper load-balancing can result in only one of the SPs out of the eight or more doing useful work while others are idle due to bad memory access patterns or divergent execution paths (if-statements) that are partially taken by the different SPs. In the cases mentioned, where the code is parallel, the actual execution is sequential. This is a major challenge with parallel algorithms in general, but is an even harder problem for GPU system.

It is very difficult to find a merging algorithm that can achieve a high level of parallelism and maximize utilization on the GPU due to the multi-level parallelism of the architecture. In a sense, sorting algorithms are easier to adapt to the GPU as the number of computations per element is larger, which allows for better utilization of the system. The algorithm that is presented in this paper uses the many cores of the GPU while reducing the number of memory requests to the global memory by using the local shared-memory in an efficient manner. We further show that the algorithm is portable for the different CUDA compute capabilities by showing the results on both TESLA and FERMI architectures. These results are also compared with an OpenMP (OMP) implementation on an x86 system.

2. GPU MERGEPATH

In this section we will present our new algorithm for the merging of two sorted arrays into a single sorted array on a GPU. Because our algorithm makes use of Merge Path [7] and the fundamental ideas behind Merge Path, we give an introduction to Merge Path for PRAM systems. We also explain why Merge Path cannot be directly implemented on a GPU and what needs to be done to port Merge Path GPU. For a deeper discussion on Merge Path we suggest reading the [7] paper.

Algorithm 1 Pseudo code for parallel merge algorithm with an emphasis on the partitioning stage. This pseudo-code most reflects the x86 implementation of Merge Path.

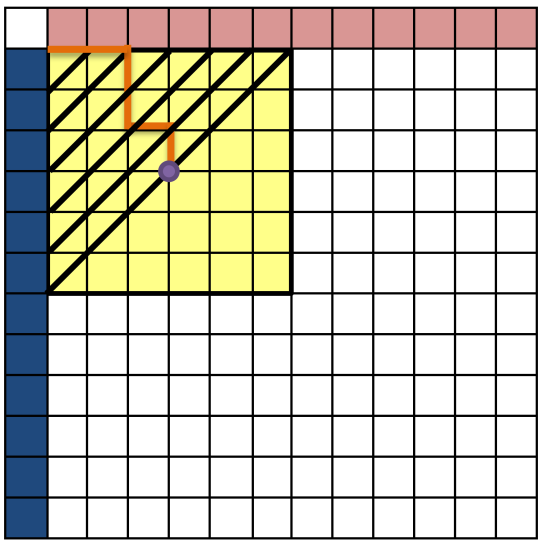
```
$A_{diag}[threads] \Leftarrow A_{length}$
$B_{diag}[threads] \Leftarrow B_{length}$
for each $i$ in $threads$ in parallel do
  $index \Leftarrow i * (A_{length} + B_{length})\ /\ threads$
  $a_{top} \Leftarrow index > A_{length}\ ?\ A_{length} : index$
  $b_{top} \Leftarrow index > A_{length}\ ?\ index - A_{length} : 0$
  $a_{bottom} \Leftarrow b_{top}$
  // binary search for diagonal intersections
  while true do
    $offset \Leftarrow (a_{top} - a_{bottom})/2$
    $a_i \Leftarrow a_{top} - offset$
    $b_i \Leftarrow b_{top} + offset$
    if $A[a_i] > B[b_i - 1]$ then
      if $A[a_i - 1] > B[b_i]$ then
        $A_{diag}[i] \Leftarrow a_i$
        $B_{diag}[i] \Leftarrow b_i$
      else
        $a_{top} \Leftarrow a_i - 1$
        $b_{top} \Leftarrow b_i + 1$
      end if
    else
      $a_{bottom} \Leftarrow a_i + 1$
    end if
  end while
end for
for each $i$ in $threads$ in parallel do
  merge($A, A_{diag}[i], B, B_{diag}[i], C, i * length/threads$)
end for
```

2.1 Parallel Merge

In [7] the authors suggest a new approach for the parallel merging of two sorted arrays on a PRAM shared-memory systems. Assuming that the system has p cores, each core is responsible for merging an equal part of the final output array. As each core receives an equal amount of work, this ensures that all the p cores will finish at the same time. Creating the balanced workload is one of the aspects that makes Merge Path a suitable candidate for porting to the GPU. While the results in [7] intersect with those in [3], the approach that is presented is different and easier to understand. We use this approach to port Merge Path to a GPU. Merge Path, like other parallel merging algorithms, is divided into 2 stages:

1. Partitioning stage - each core is responsible for dividing the input arrays into partitions. Each core will find a single non-overlapping partition in each of the input arrays. While the sub-arrays of each partition are not equal length, the sum of the lengths of the two sub-arrays of a specific partitions will be equal (up to a constant of 1) among all the cores. We present the pseudo code for the partitioning and give a brief explanation. For more information we suggest reading [7, 3] .

2. Merging stage - each core merges the two sub arrays that it has been given using a sequential merge. The cores operate on non-overlapping segments of the out-


(a) Initial position of the window which.

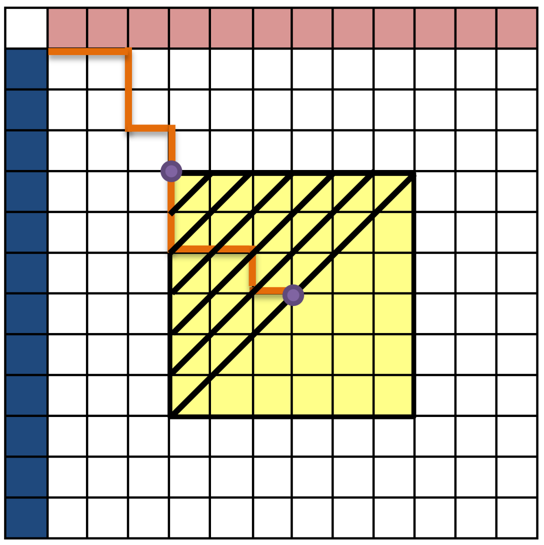

(b) New position of window(after completion of previous block).

Figure 2: Diagonal searches for a single window of one MP. (a) The window is in its initial position. All the SPs do a search for the path. (b) The window is move to the furthermost position of the path and the diagonals are searched a second time.

put array, thus the merging can be done concurrently and lock-free.

It is worth noting that both of these stages are parallel.

Merge Path suggests treating the sequential merge as if it was a path that moves from the top-left corner of a rectangle to the bottom-right corner of the rectangle $(|B|, |A|)$. The path can move only to the right and down. We denote $n = |A + |B|$. Consequently, when the entire path is known, so is the order in which the merge takes place. Thus, when an entire path is given, it is possible to complete the merge concurrently; however, at the beginning of the merge the path is unknown and computation of the entire path is considerably expensive $O(n \cdot log(n))$ compared with the complexity of sequential merge which is $O(n)$. Therefore, we will compute only p points on this path and these points will be the partitioning points. For the sake of brevity, we present only the idea of how to find the partitioning points on the path without the formal proofs. The formal proofs can be found in [7]. The main idea of finding the partitioning points is to use cross diagonals that start at the top and right borders of the output matrix. The cross diagonals are bound to meet the path at some point as the path moves from the top-left to the bottom-right and the cross diagonals move from the top-right to the bottom-left. It is possible to find the points of intersection using a binary search on the cross diagonals to find the point at which the '1's and '0's meet by comparing elements from A and B, making the complexity of finding the intersection $O(log(n))$. By finding exactly p points on the path such that these points are equally distanced, we ensure that the merge stage is perfectly load balanced. By using equidistant cross diagonals, the work load is divided equally among the cores, see [7]. The time complexity of this algorithm is $O(log(n) + n/p)$. This is also the work load of each of the processors in the system. The work complexity of this algorithm is $O(p \cdot log(n) + n)$.

Sequential merging of the sub-arrays is the same as the a regular sequential merge that was presented earlier in this paper. The pseudo code for this algorithm is presented in Algorithm 1.

2.2 GPU Partitioning

The above parallel selection algorithm can be easily implemented on a PRAM machine as each core is responsible for a single diagonal. This approach can be implemented on the GPU; however, it will suffer from low utilization as only one SP for each SM is being utilized at any given and the rest are idle. We will present 3 approaches to implement the cross diagonal binary search followed by a detailed description. We denote w as the size of the warp. The 3 approaches are:

1. w-wide binary search.
2. Regular binary search.
3. w-partition search.

Detailed description of the approaches is as follows:

1. w-wide binary search - in this approach we fetch w consecutive elements from each of the arrays A and B. By using CUDA block of size w, each SP/thread is responsible for fetching a single element from the global array, which is in the global memory, into the shared-memory. This is efficient use of the memory system

on the GPU as the addresses are consecutive, thus incurring a single memory request. As the intersection is a single point, only one SP can find the intersection and store the point of intersection in global memory, which removes the need for synchronization. It is rather obvious that the complexity of this search is greater than the one presented in Merge Path[7] which does a regular sequential search for each of the diagonals; however, the GPU architecture requires that the minimal number of threads of a block be at the size of a warp, doing w searches or doing 1 search takes the same amount of time in practice as the additional execution units would otherwise be idle. In addition to this, the GPU architecture has a wide memory bus that can bring more than a single data element per cycle making it cost wise effective to use the fetched data. In essence for each of the stages in the binary search, a total of w operations are completed. This approach reduces the number of searches required by a multiple of $log(w)$. The complexity of this approach for each diagonal is: $Work = O(w \cdot log(N) - log(w))$ and $Time = O(log(N) - log(w))$ for each core.

2. Regular binary search - simply a single-threaded binary search on each diagonal. The complexity of this: $Work = O(log(N))$ and $Time = O(log(N)))$. In actuality the work is the same as in the w-wide binary search as the cores in the warp are idle but are still doing some background operations.

3. w-partition search - in this approach, the cross diagonal is divided into 32 equal-size and independent partitions. Each thread in the warp is responsible for a single comparison. Each thread checks to see if the point of intersection is in its partition. Similar to w-wide binary search, there can only be one partition that the intersection point goes through. Therefore, no synchronization is needed. An additional advantage of this approach is that in each iteration the search space is reduced by a factor of w rather then the 2 as in binary search. This reduces the $O(log(N))$ comparisons needed to to $O(log_w(N))$ comparisons. The complexity of this approach for each diagonal is: $Work = w \cdot log_w(N) - log(w)$ and $Time = log_w(N) - log(w)$. The biggest shortcoming of this approach is that for each iteration of that partition-search, a total of w memory requests are needed(one for each of the partitions limits). As the memory requests fall on different lines, the implementation suffers a performance penalty waiting on memory requests to complete.

In conclusion, we tested all of these approaches, and the first two approaches offer a significant performance improvement over the last due to a reduced number of memory requests to the global memory for each iteration of the search. This is especially important as the computation time for each iteration of the searches is considerably smaller than the memory latency. The time difference between the first two approaches is negligible with a slight advantage to one or the other depending on the input data. For the results that are presented in this paper we use the w-wide binary search.

2.3 GPU Merge

The merge phase of the Merge Path algorithm is not well-suited for the GPU as the merging stage is purely sequential for each core. Therefore, it is necessary to find a way to parallelize the merge that will utilize all the SPs on a single SM once the partitioning stage is completed.

For full utilization of the SMs in the system, the merge must be broken up into finer granularity to enable additional parallelism while still avoiding synchronization when possible. We present our approach on dividing the work among the multiple SPs for a specific workload.

For the sake of simplicity assume that the sizes of the partitions that are created by the partition stage are significantly greater than the warp size, w. Also we will denote the CUDA thread block size using the letter Z and assume that it too is greater or equal to w. For practical purposes $Z = 32$ or 64; however, anything that is presented in this subsection can also be used with larger Z. Take the Z smallest elements of each of the partitions and place them in shared memory (in a sequential manner for performance benefit). Z is smaller than the shared-memory and therefore can fit the data. Using a Z thread block, it is possible to find the exact Merge Path of the Z elements using the cross diagonal binary search. Given the full path for the Z elements it is possible to know how to merge all the Z elements concurrently as each of the elements will be written to a specific index. The complexity for finding the entire Z-length path requires $log(Z)$ in general iterations. This is followed by placing the elements in their respective place in the output array. Upon completion of the Z-element merge, it is possible to move on to unmerged elements by updating the 'top-left' such that it is the value of the 'bottom-right' corner of the previous Merge Path. This can be seen in Fig. 2 where the window starts off at the initial point of merge for a given SM. All the threads do a diagonal search looking for the path. Moving the window is a simple operation as it requires only moving the pointers of the sub-arrays according to the (x, y) lengths of the path. This operation will be repeated until the SM finishes merging the two sub-arrays that it was given. The only performance requirement of the algorithm is that the sub-arrays fit into the shared-memory of the SM. If the sub-arrays fit in the shared-memory, the SPs can perform random memory access without incurring significant performance penalty. To further offset the overhead of the path searching, we will let each of the SPs merge several elements. In other words, this means that for a Z threads, we will merge $\alpha \cdot Z$ elements for some $\alpha > 1$ and $\alpha \in \mathbb{N}$.

2.4 Complexity analysis of the GPU merge

Given p blocks of Z threads and a n elements to merge where n is the total size of the output array, the size of the partition that each of the blocks of threads will receive is in n/p. Following the explanation in the previous subsection on the movement of the sliding window, the window will move a total of $(n/p)/Z$ times for that window. For each window, each thread in the block performs a binary diagonal search that is dependent on the block size Z. When the search is complete, the threads copy their resulting elements into independent locations in the output array directly. Thus, the time complexity of merging a single window is $O(log(Z))$. The total amount of work that is completed for a single block is $O(Z \cdot log(Z))$. The total time complexity for the merging done by a single thread block

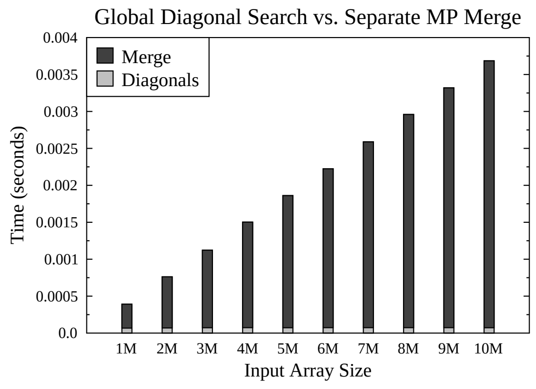


Figure 3: The timing of the global diagonal search to divide work among the MPs compared with the timing of the independent parallel merges performed by the MPs.

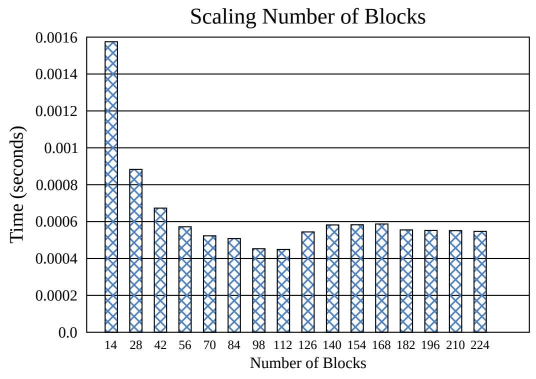


Figure 4: Merging one million single-precision floating point numbers in Merge Path on Fermi while varying the number of thread blocks used from 14 to 224 in multiples of 14.

and thus all blocks in parallel is $O((n/p)/Z \cdot log(Z))$ and the work complexity of the entire merge is $O(n \cdot log(Z))$

2.5 GPU Optimizations and Issues

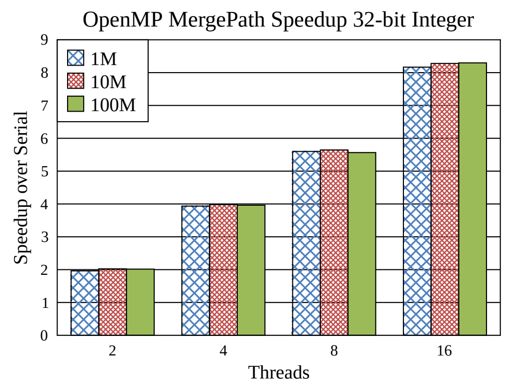
1. Memory transfer between global and shared memory is done in sequential reads and writes. A key requirement for the older versions of CUDA, versions 1.3 and down, is that when the global memory is accessed by the SPs, the elements requested be co-located and not permuted. If the accesses are not sequential, the number of memory requests increases and the entire warp (or partial-warp) is frozen until the completion of all the memory requests. Acknowledging this requirement and making the required changes to the algorithm has allowed for a reduction in the number of memory requests to the global memory. In the shared memory it is possible to access the data in a non-sequential fashion without as significant of a performance degradation.
2. Assume the existence of virtual cores - this approach suggests that more diagonals are computed than the number of SMs in the system. This increases performance for both of the stages in the algorithm. This is due to the internal scheduling of the GPU which requires a large number of threads and blocks to achieve the maximal performance through latency hiding.

3. EMPIRICAL RESULTS

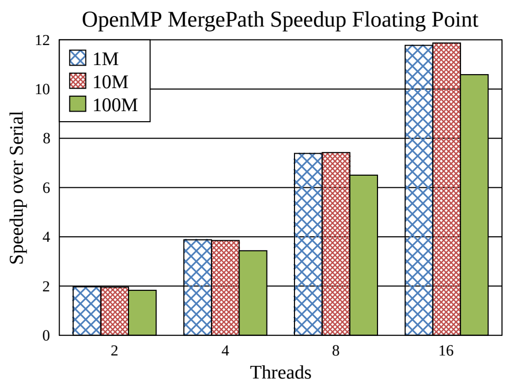
In this section we present comparisons of the running time of the GPU Merge Path algorithm on the NVIDIA Tesla and Fermi GPU architectures with those of Merge Path on a x86 Intel Nehalem core system. For the x86 system we show results for both a sequential merge and for OpenMP implementation of Merge Path. In addition to this, we compare GPU Merge path with the merge algorithm supplied by the Thrust library[5]. Thrust is a parallel primitives library that is included in the default installation of the CUDA SDK Toolkit as of version 4.0. The specifications of the GPUs used can be seen in Table 1, and the Intel multi-core configuration can be seen in Table 2. An efficient sequential merge is used to provide a basis for comparison. Tests were run for input array sizes of one million, ten million, and one hundred million elements for a merged array size of two million, twenty million, and two hundred million merged elements respectively. For each size of array, merging was tested on both single-precision floating point numbers and 32-bit integers. Our results demonstrate that the GPU architecture can achieve a 2x to 5x speedup over using 16 threads on two hyper-threaded quad-core processors. This is an effect of both the greater processing power and higher memory bandwidth of the GPU architecture.

Initially, we ran tests to obtain an optimal number of thread blocks to use on the GPU for best performance with separate tests for Tesla and Fermi. Results for this block scaling test on Fermi at one million single-precision floating point numbers using blocks of 128 threads can be seen in Fig. 4. Clearly, the figure shows the best performance is achieved at 112 blocks of 128 threads for this particular case. Similarly, 112 blocks of 128 threads also produces the best results in the case of merging one million 32-bit integer elements on Fermi. For the sake of brevity, we do not present the graph.

The GPUs used were a Tesla C1060 supporting CUDA hardware version 1.3 and a Tesla C2070 (Fermi architecture) supporting CUDA hardware version 2.0. The primary differences between the older Tesla architecture and the newer Fermi architecture are the increased size of the shared memory per SM from 16KB to 64KB, the option of using all or some portion of the L1 shared memory as a hardware-controlled cache, increased total CUDA core count, increased memory bandwidth, and increased SM size from 8 cores to 32 cores. In our experiments, we use the full shared memory configuration. Managing the workspace of a thread block in software gave better results than allowing the cache replacement policy to control which sections of each array were in the shared memory. Larger shared memory and increased core counts allow the size of a single window on the Fermi architecture to be larger with each block of 128 threads merging 4 elements per window for a total of 512 merged-elements per window. The thread block reads 512 elements cooperatively from each array into shared memory, perform the Merge Path algorithm operating on this smaller problem

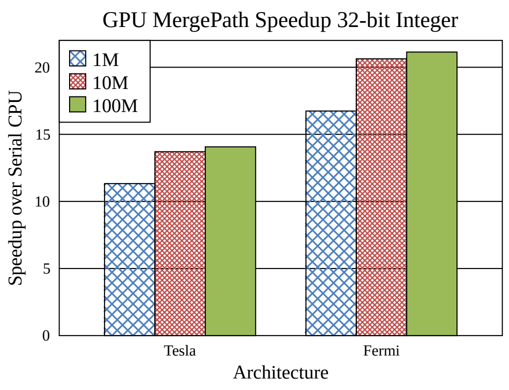


(a) 32-bit integer merging in OpenMP

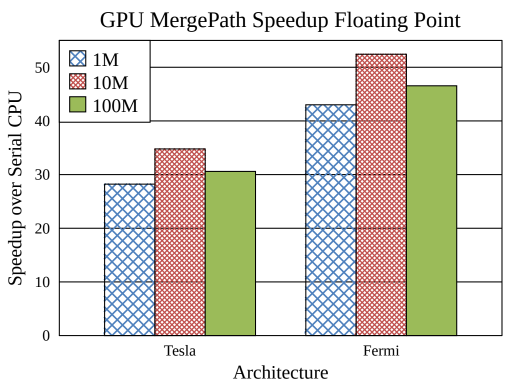


(b) Single-precision floating point merging in OpenMP

Figure 5: The speedup of the OpenMP implementation of Merge Path on two hyper-threaded quad-core Intel Nehalem Xeon E5530s over serial merge.

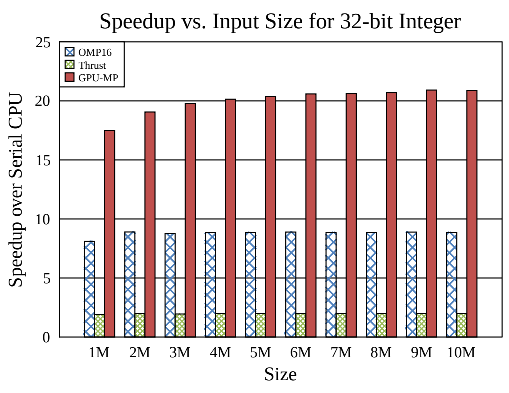


(a) 32-bit integer merging on GPU

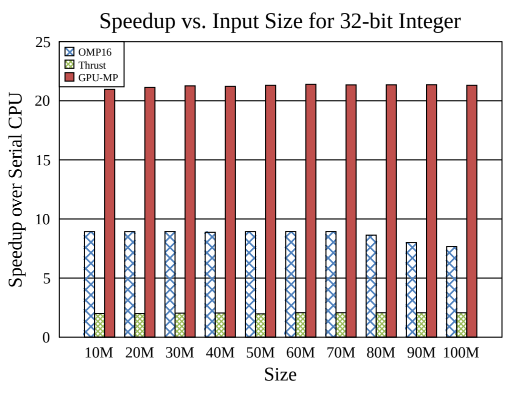


(b) Single-precision floating point merging on GPU

Figure 6: The speedup of the GPU implementations of Merge Path on the NVIDIA Tesla and Fermi architectures over serial merge.

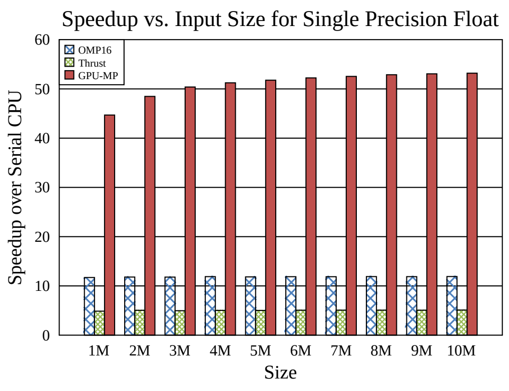


(a) 1M to 10M elements

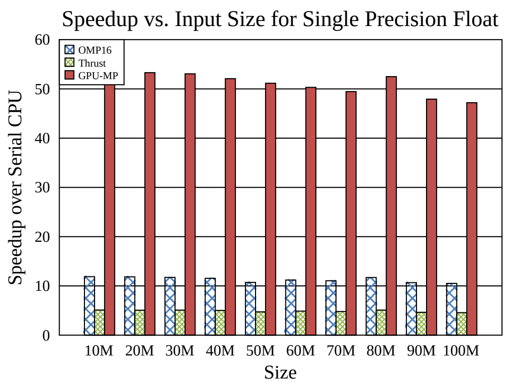


(b) 10M to 100M elements

Figure 7: Comparison of merging 32-bit integers using Merge Path on 16 OpenMP threads, Merge Path on 112 blocks of 128 threads on Fermi, and thrust::merge on Fermi. Size refers to the length of a single input array in elements.



(a) 1M to 10M elements



(b) 10M to 100M elements

Figure 8: Comparison of merging single-precision floating point numbers using Merge Path on 16 OpenMP threads, Merge Path on 112 blocks of 128 threads on Fermi, and thrust::merge on Fermi. Size refers to the length of a single input array in elements.

Table 1: GPU test-bench.

Card Type	CUDA HW	CUDA Cores	Frequency	Shared Memory	Global Memory	Memory Bandwidth
Tesla C1060	1.3	240 cores / 30 SMs	1.3 GHz	16KB	2GB	102 GB/s
Fermi M2070	2.0	448 cores / 14 SMs	1.15 GHz	64KB	6GB	150.3 GB/s

Table 2: CPU test-bench.

Processor	Cores	Clock Frequency	L2 Cache	L3 Cache	DRAM	Memory Bandwidth
2 x Intel Xeon E5530	4	2.4GHz	4 x 256 KB	8MB	12GB	25.6 GB/s

out of shared memory, then cooperatively write back 512 elements. In the Tesla implementation, only 32 threads per block are used to merge 4 elements per window for a total of 128 merged-elements due to shared memory size limitations and memory latencies. Speedup for the GPU implementation on both architectures are presented in Fig. 6.

As the GPU implementations are benchmarked in comparison to the x86 sequential implementation, we first present these results. This is followed by the results of the GPU implementation. The timings for the sequential and OpenMP implementations are performed on a machine presented in Table 1. For the OpenMP timings we run the Merge Path algorithm using 2, 4, 8, and 16 threads on an 8-core system that supports hyper-threading. As hyper-threading requires two threads per core to share execution units and additional hardware, the performance per thread is reduced versus two threads on independent cores. The hyper-threading option is used only for the 16 thread configuration. For the 4 thread configuration we use a single quad core socket, and for the 8 thread configuration we use both quad core sockets. For each configuration, we perform three merges of two arrays of sizes 1 million, 10 million, and 100 million elements as with the GPU. The speedups are presented in Fig. 5. Within a single socket, we see a linear speedup to four cores. Going out of socket, the speedup for eight cores was 5.5X for integer merging and between 6X and 8X for floating point. Our results are inconsistent with those presented in [7]. We explain this by the fact that we use a different system and architecture. This can be attributed to the limitations of the QPI interconnect compared to the on-chip network. Additionally, the work of each thread is reduced while the overhead of OpenMP stays the same (or possibly increases due to additional thread creations) making the overhead a more significant part of the work. Regardless, we show a 12x speedup for the hyper-threading configuration for ten million elements.

The main focus of our work was to show that Merge Path can be adapted to work on GPUs. We now present the speedup results of the GPU Merge Path over the sequential merge in Fig. 6. We use the same size arrays for both the GPU and OpenMP implementations. For our results we use equal size arrays for merging, however, our method can merge arrays of different sizes. In Fig. 3, we show a comparison of the runtime of the two kernels: partitioning and merging. It can be seen that the partitioning stage has a negligible runtime compared with the actual parallel merge operations and increases in runtime only very slightly with increased problem size, as expected. This chart also demonstrates that our implementation scales linearly in runtime with problem size while utilizing the same number of cores.

3.1 Floating Point Merging

For the floating point tests, random floating point numbers were generated to fill the input arrays then sorted on the host CPU. The input arrays were then copied into the GPU global memory. These steps are not timed as they are not relevant to the merge. The partitioning kernel is called first. When this kernel completed a second kernel is called to perform full merges between the global diagonal intersections using parallel merge path windows per thread block on each SM was called. These kernels were timed. For the arrays of size 1 million, 10 million, 100 millions arrays we see a significant speedup of 28X & 34X & 30X on the Tesla card and 43X & 53X & 46X on the Fermi card, this is depicted in Fig. 6. In Fig. 8, we directly compare the speedup of the fastest GPU (Fermi) implementation, the 16-thread OpenMP implementation, and the Thrust GPU merge implementation demonstrating that the GPU code ran nearly 5x faster than 16-threaded OpenMP and nearly 10x faster than Thrust merge for floating points operations. The speedup of our algorithm over OpenMP is a direct effect of the difference in memory bandwidth of the two platforms.

3.2 Integer Merging

The integers speedup on the GPU are depicted in Fig. 6 for arrays of size 1 million, 10 million, 100 millions arrays we see a significant speedup of 11X & 13X & 14X on the Tesla card and 16 & 20X & 21X on the Fermi card. In Fig. 7, we directly compare the speedup over serial of the fastest GPU (Fermi) implementation, the 16-thread OpenMP implementation, and the Thrust GPU merge implementation demonstrating that the GPU code ran nearly 2.5X faster than 16-threaded OpenMP and nearly 10x faster on average than the Thrust merge. The number of blocks used for Fig. 7 is 112 blocks, similar to the number of blocks used in the floating point sub-section.

4. CONCLUSION

In the previous sections the results for integers and floating points were presented. These results differ in the speedup achieved with each of the types; however, it is worth noting that execution time for merging for both these types on the GPU are nearly the same. The explanation for this phenomena is that the execution time for integers on the CPU is 2.5X faster than the execution time for floating points on the CPU. This explains the reduction in the the speedup of integers on the GPU in comparison with speedup of the floating point.

We have shown in this paper a way to implement an efficient atomic free parallel merging of two arrays using a GPU. The algorithm is scalable and adaptable to the different compute capabilities and architectures that are provided NVidia. In our benchmarking, we show that the new algorithm outperforms a sequential merge by more than an order of magnitude and outperforms an OpenMP implementation of Merge Path that uses 8 hyper-threaded cores by a factor of 2.5X. This new approach uses the GPU efficiently and takes advantage of the computational power of the GPU and memory system by using the global memory, shared-memory and the bus of the GPU effectively. This new algorithm would be beneficial for many GPU sorting algorithms including for a GPU merge sort algorithm.

5. ACKNOWLEDGMENTS

We would like to acknowledge Saher Odeh and Yitzhak Birk for their discussion and comments.

6. REFERENCES

[1] S. Chen, J. Qin, Y. Xie, J. Zhao, and P. Heng. An efficient sorting algorithm with cuda. *Journal of the Chinese Institute of Engineers*, 32(7):915–921, 2009.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[3] N. Deo, A. Jain, and M. Medidi. An optimal parallel algorithm for merging using multiselection. *Information Processing Letters*, 1994.

[4] N. K. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[5] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[6] NVIDIA Corporation. Nvidia cuda programming guide. 2011.

[7] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - cache-efficient parallel merge and sort. Technical report, CCIT Report No. 802, EE Pub. No. 1759, Electrical Engr. Dept., Technion, Israel, Jan. 2012.

[8] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

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

[10] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381 – 1388, 2008. General-Purpose Processing using Graphics Processing Units.