GPU Merge Path - A GPU Merging Algorithm

Oded Green *
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
ogreen@gatech.edu

Robert McColl
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
rmccoll3@gatech.edu

David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
bader@cc.gatech.edu

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 set intersection. An efficient parallel merging algorithm partitions the sorted input arrays into sets of non-overlapping sub-arrays that 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 partitions the workload equally amongst the GPU Streaming Multiprocessors (SM). Following this, we show how each SM performs a parallel merge and how to divide the work so that all the GPU's Streaming Processors (SP) are utilized. All stages in this algorithm are parallel. The new algorithm demonstrates good utilization of the GPU memory hierarchy. This approach demonstrates an average of 20X and 50X speedup over a sequential merge on the x86 platform for integer and floating point, respectively. Our implementation is 10X faster than the fast parallel merge supplied in the CUDA Thrust library.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

*Corresponding author

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

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. Since merging is a common primitive in larger applications, improving performance through parallel computing approaches can provide benefit to existing codes used in a variety of 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]. As of now, it will be assume that $|C| = n$.

The increase in the number of cores in modern computing systems presents an opportunity to improve performance through clever parallel algorithm design; however, there are numerous challenges that need to be addressed for a parallel algorithm to achieve optimal performance. These challenges include evenly partitioning the workload for effective load balancing, reducing the need for synchronization mechanisms, and minimizing the number of redundant operations caused by the parallelization. We present an algorithm that meets all these challenges and more for 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. In particular, we present Merge Path [8, 7]. Section 2 introduces our new GPU merging algorithm, GPU Merge Path, and explains the different granularities of parallelism present in the algorithm. In section 3, we show empirical results of the new algorithm on two different GPU architectures and improved performance over existing algorithms on GPU and x86. Section 4 offers concluding remarks.

1.1 Introduction on GPU

Graphics Processing Units (GPUs) have become a popular platform for parallel computation in recent years following the introduction of programmable graphics architectures like NVIDIA's Compute Unified Device Architecture (CUDA) [6] that allow for easy utilization of the cards for purposes other than graphics rendering. For the sake brevity, we present only a short introduction to the CUDA architecture.

To the authors' knowledge, the parallel merge in the Thrust

library [5] is the only other parallel merge algorithm implemented on a CUDA device.

The original purpose of the GPU is 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 are also known as stream processors (SP), and they are arranged into groups of 8/16/32 (depending on the CUDA compute capability) cores known as stream multiprocessors (SM). The exact number of SMs on the card is dependent on the particular model. Each SM has a single control unit responsible for fetching and decoding instructions. All the SPs for a single SM execute the same instruction at a given time but on different data or perform no operation in that cycle. Thus, the true concurrency is limited by the number of physical SPs. The SMs are responsible for scheduling the threads to the SPs. The threads are executed in groups called warps. The current size of a warp is 32 threads. Each SM has a local shared memory / private cache. Older generation GPU systems have 8KB local shared memory, whereas the new generation has 64KB of local shared memory which can be used in two separate modes.

In CUDA, users must group threads into blocks and construct a grid of some number of thread blocks. The user specifies the number of threads in a block and the number of blocks in a grid that will all run the same kernel code. Kernels are functions defined by the user to be run on the device. These kernels may refer to a thread's index within a block and the current block's index within the grid. A block will be executed on a single SM. For full utilization of the SM it is good practice to set the block size to be a multiple of the warp. As each thread block is executed by a single SM, the threads in a block can share data using the local shared memory of the SM. A thread block is considered complete when the execution of all 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

The focus of this paper is on parallel merging; however, there has not been significant study solely on parallel merging on the GPU. Therefore we give a brief description of prior work in the area of sorting: sequential, multicore parallel sorting, and GPU parallel sorting [9, 11]. In further sections there is 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, 8, 10] . Following the GPGPU trend, several algorithms have been suggested that implement sorting using a GPU for increased performance. For additional reading on parallel sorting algorithms and intricacies of the GPU architecture (specifcally NVIDIA's CUDA), we refer the reader to [9, 4, 1] on GPU sorting.

Some of the new algorithms are based on a single sorting method such as the radix sort in [9]. In [9], Satish et al. 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

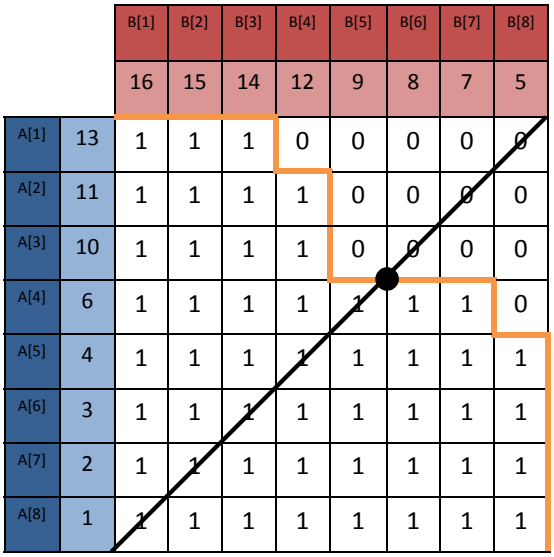


Figure 1: Illustration of the MergePath concept for a non-increasing merge. The first column (in blue) is the sorted array A and the first row (in red) is the sorted array B. The orange path (a.k.a. Merge Path) represents comparison decisions that are made to form the merged output array. The black cross diagonal intersects with the path at the midpoint of the path which corresponds to the median of the output array.

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 the one presented by Sintorn and Assarsson [11], 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 Merging and Merge Path

While serial merging follows 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. Finally, copy the remaining elements from the other input array into the end of the output array [2].

In [8], the authors suggests treating the sequential merge as though it is 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 this way, performing a merge can be thought of as the process of discovering this path through a series of comparisons. When $A[i] \geq B[j]$, the path moves down by one position, and we copy $A[i]$ into the appropriate place in C. Otherwise when $A[i] < B[j]$, the path moves to the right, and we copy $B[j]$ into the appropriate place in C. We can determine the path directly by doing comparisons. Similarly, if we determine a point on the path through a means other than doing all comparisons that lead to that point, we have determined something about the outcomes of those comparisons earlier in the path. From a pseudo-code point of view, when the path moves to the right, it can be considered taking the branch of the condition when $A[i] \leq B[j]$, and when the path moves down, it can be thought of as not taking that branch. Consequently, given this path the order in which the merge is to be completed is totally known. Thus, all the elements can be placed in the output array in parallel based on the position of the corresponding segment in the path (where the position in the output array is the sum of row and column indices of a segment).

In this section we address only the sequential version of Merge Path. In the following sections we further discuss the parallelization of Merge Path on an x86 architecture and its GPU counterpart.

We can observe that sections of the path correspond to sections of elements from at least one or both of the input arrays and a section of elements in the output array. Here each section is distinct and contiguous. Additionally, the relative order of these workspaces in the input and output arrays can be determined by the relative order of their corresponding path sections within the overall path. In Fig. 1, $A[1] = 13$ is greater than $B[4] = 12$. Thus, it is greater than all $B[j]$ for $j \geq 4$. We mark these elements with a '0' in the matrix. This translates to marking all the elements in the first row and to the right of $B[4]$(and including) with a '0'. In general if $A[i] > B[j]$, then $A[i'] > B[j]$ for all $i' \leq i$. 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'.

Observation 1: The path that follows along the boundary between the '0's and '1's is the same path mentioned above which represents the selections from the input arrays that form the merge as is depicted in Fig. 1 with the orange, 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).

Observation 2: Paths can only move down and to the right from a point on one diagonal to a point on the next. Therefore, if the cross diagonals are equally spaced diagonals, the lengths of paths connecting each pair of cross diagonals are equal.

1.4 GPU Challanges

To achieve maximal speedup on the GPU platform, it is necessary to implement a platform-specific (and in some cases, card-specific) algorithm. These implementations are architecture-dependent and in many cases require a deep understanding of the memory system and the execution system. Ignoring the architecture limits the achievable performance. For example, a well known performance hinderer is bad memory access (read/write) patterns to the global memory. Further, GPU-based applications greatly benefit from implementation of algorithms that are cache aware.

For good performance, all the SPs on a single SM should read/write sequentially. If the data is not sequential (meaning that it strides across memory lines in the global DRAM), this could lead to multiple global memory requests which cause all SPs to wait for all memory requests to complete. One way to achieve efficient global memory use when non-sequential access is required is to do a sequential data read into the local shared memory incurring one memory request to the global memory and followed by 'random' (non-sequential) memory accesses to the local shared memory.

An additional challenge is to find a way to divide the workload evenly among the SMs and further partition the workload evenly among 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 global memory access patterns or divergent execution paths (if-statements) that are partially taken by the different SPs. For the cases mentioned, where the code is parallel, the actual execution is sequential.

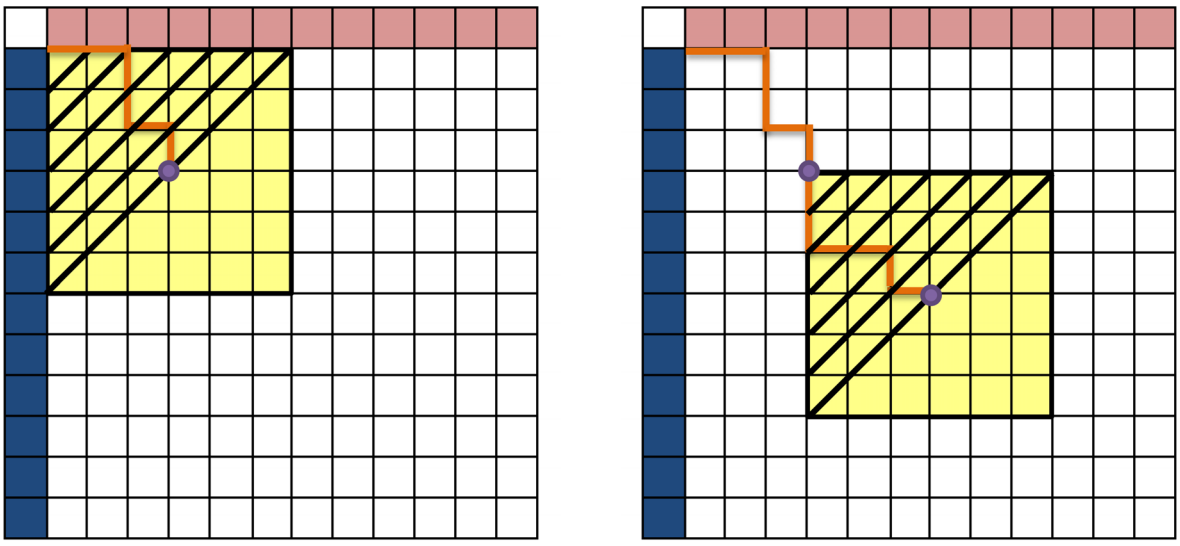
It is very difficult to find a merging algorithm that can achieve a high level of parallelism and maximize utilization on the GPU due to the multi-level parallelism requirements of the architecture. In a sense, parallelizing merging algorithms is even more difficult due to the small amount of work done per each element in the input and output. The algorithm that is presented in this paper uses the many cores of the GPU while reducing the number of requests to the global memory by using the local shared memory in an efficient manner. We further show that the algorithm is portable for different CUDA compute capabilities by showing the results on both TESLA and FERMI architectures. These results are compared with the parallel merge from the Thrust library on the same architectures and an OpenMP (OMP) implementation on an x86 system.

2. GPU MERGEPATH

In this section we present our new algorithm for the merging of two sorted arrays into a single sorted array on a GPU. In the previous section we explained Merge Path [8] and its key properties. In this section, we give an introduction to parallel Merge Path for parallel systems. We also explain why the original Merge Path algorithm cannot be directly implemented on a GPU and our contribution development of the GPU Merge Path algorithm.

2.1 Parallel Merge

In [8] the authors suggest a new approach for the parallel merging of two sorted arrays on parallel shared-memory systems. Assuming that the system has p cores, each core is responsible for merging an equal n/p part of the final output array. As each core receives an equal amount of work, this ensures that all the p cores finish at the same time. Creating the balanced workload is one of the aspects that makes Merge Path a suitable candidate for the GPU. While the results in [8] intersect with those in [3], the approach that is presented is different and easier to understand. We use this approach to develop a Merge Path algorithm for



(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 SM. (a) The window is in its initial position. Each SP does a search for the path. (b) The window moves to the farthest position of the path and the new diagonals are searched.

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 finds 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 partition is equal (up to a constant of 1) among all the cores. In Algorithm 1 and in the next section, we present the pseudo code for the partitioning and give a brief explanation. For more information we suggest reading [3, 7, 8] .

2. Merging stage - Each core merges the two sub arrays that it has been given using the same algorithm as a simple sequential merge. The cores operate on non-overlapping segments of the output array, thus the merging can be done concurrently and lock-free. Using the simple sequential mere algorithm for this stage is not well-suited to the GPU.

Both of these stages are parallel.

As previously stated, 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 $n = |A+|B|$. 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 through a diagonal binary search for all the cross diagonals is considerably expensive $O(n \cdot log(n))$ compared with the complexity of sequential merge which is $O(n)$. Therefore, we compute only p points on this path. These points are the partitioning points.

To find the partitioning points, 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 binary searches on the cross diagonals by comparing elements from A and B (Observation 1), 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 [8]. The merging of the sub-arrays in each partition is the same as the standard sequential merge that was discussed earlier in this paper. The time complexity of this algorithm is $O(n/p + log(n))$. This is also the work load of each of the processors in the system. The work complexity of this algorithm is $O(n + p \cdot log(n))$.

2.2 GPU Partitioning

The above parallel selection algorithm can be easily implemented on a parallel machine as each core is responsible for a single diagonal. This approach can be implemented on the GPU; however, it does not fully utilize the GPU as SPs on each SM will frequently be idle. We present 3 approaches to implementing the cross diagonal binary search

Algorithm 1 Pseudo code for parallel Merge Path algorithm with an emphasis on the partitioning stage.

```
A_diag[threads] ⇐ A_length
B_diag[threads] ⇐ B_length
for each i in threads in parallel do
  index ⇐ i * (A_length + B_length) / threads
  a_top ⇐ index > A_length ? A_length : index
  b_top ⇐ index > A_length ? index − A_length : 0
  a_bottom ⇐ b_top
  // binary search for diagonal intersections
  while true do
    offset ⇐ (a_top − a_bottom)/2
    a_i ⇐ a_top − offset
    b_i ⇐ b_top + offset
    if A[a_i] > B[b_i − 1] then
      if A[a_i − 1] ≤ B[b_i] then
        A_diag[i] ⇐ a_i
        B_diag[i] ⇐ b_i
        break
      else
        a_top ⇐ a_i − 1
        b_top ⇐ b_i + 1
      end if
    else
      a_bottom ⇐ 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
```

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 each of the global arrays, which are in the global memory, into the local shared memory. This efficiently uses the memory system on the GPU as the addresses are consecutive, thus incurring a minimal number of global memory requests. As the intersection is a single point, only one SP finds the intersection and stores the point of intersection in global memory, which removes the need for synchronization. It is rather obvious that the work complexity of this search is greater than the one presented in Merge Path[8] which does a regular sequential search for each of the diagonals; however, doing w searches or doing 1 search takes the same amount of time in practice as the additional execution units would otherwise be idle if we were executing only a single search. 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-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 factor of w. The complexity of this approach for each diagonal is: $Time = O(log(n) - log(w))$ for each core and a total of $Work = O(w \cdot log(n) - log(w))$.

2. Regular binary search - This is simply a single-threaded binary search on each diagonal. The complexity of this: $Time = O(log(n)))$ and $Work = O(log(n))$. Note that the SM utilization is low for this case, meaning that all cores but one will idle and the potential extra computing power is wasted.

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 than 2 as in binary search. This reduces the $O(log(n))$ comparisons needed to $O(log_w(n))$ comparisons. The complexity of this approach for each diagonal is: $Time = O(log_w(n) - log(w))$ and $Work = O(w \cdot log_w(n) - log(w))$. The biggest shortcoming of this approach is that for each iteration of that partition-search, a total of w global memory requests are needed(one for each of the partitions limits). As a result, the implementation suffers a performance penalty from waiting on global 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 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 global 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 original 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 extend the algorithm to parallelize the merge stage in a way that still uses all the SPs on each 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 partitioning stage are significantly greater than the warp size, w. Also we denote the CUDA thread block size using the letter Z and assume

that $Z \geq w$. For practical purposes $Z = 32$ or 64; however, anything that is presented in this subsection can also be used with larger Z. Take a window consisting of the Z largest elements of each of the partitions and place them in local shared memory (in a sequential manner for performance benefit). Z is smaller than the local shared memory, and therefore the data will fit. 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 are written to a specific index. The complexity for finding the entire Z-length path requires $O(log(Z))$ time 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 starting a new merge window whose top-left corner starts at the bottom-right-most position of the merge path in the previous window. 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 is 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 local shared memory of the SM. If the sub-arrays fit in the local shared memory, the SPs can perform random memory access without incurring significant performance penalty. To further offset the overhead of the path searching, we let each of the SPs merge several elements.

2.4 Complexity analysis of the GPU merge

Given p blocks of Z threads and 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 receives is n/p. Following the explanation in the previous sub-section on the movement of the sliding window, the window moves a total of $(n/p)/Z$ times for that partition. 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 is $O(n/(p \cdot Z) \cdot log(Z))$ and the work complexity is $O(n/p \cdot log(Z))$. For the entire merge the time complexity stays the same, $O(n/(p \cdot Z) \cdot log(Z))$, as all the cores are expected to complete at the same time. The work complexity of the entire merge is $O(n \cdot log(Z))$.

The complexity bound given for the GPU algorithm is different than the one given in [8, 7] for the cache efficient Merge Path. The time complexity given by Odeh et el. is $O(n/p + n/\hat{Z} \cdot \hat{Z})$, where $\hat{Z}$ refers to the size of the shared memory and not the block size. It is worth noting that the GPU algorithm is also limited by the size of the shared memory that each of the SMs has, meaning that Z is bounded by the size of the shared memory.

While the GPU algorithm has a higher complexity bound, we will show in the results section that the GPU algorithm offers significant speedups over the parallel multicore algorithm.

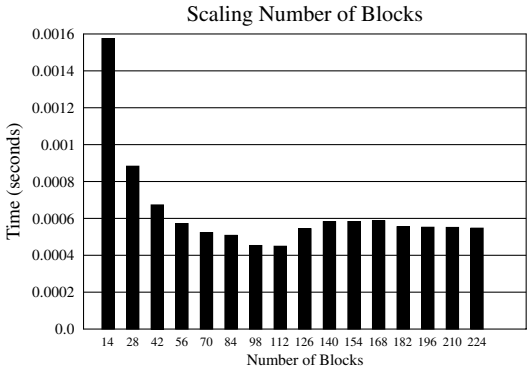


Figure 3: Merging one million single-precision floating point numbers in Merge Path on Fermi while varying the number of thread blocks used from 14 to 224 in multiples of 14.

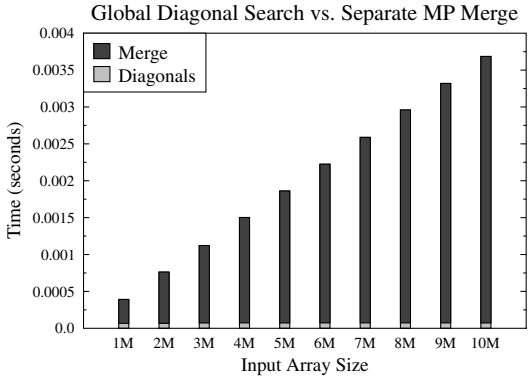
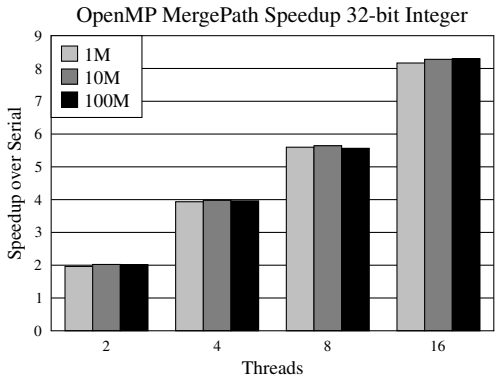


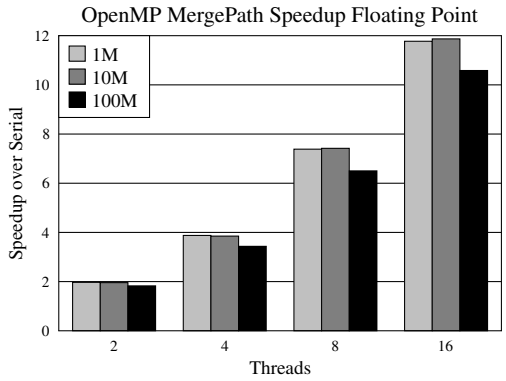
Figure 4: The timing of the global diagonal search to divide work among the SMs compared with the timing of the independent parallel merges performed by the SMs.

2.5 GPU Optimizations and Issues

1. Memory transfer between global and local 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 global 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 requests to the global memory. In the local shared memory it is possible to access the data in a non-sequential fashion without as significant of a performance degradation.

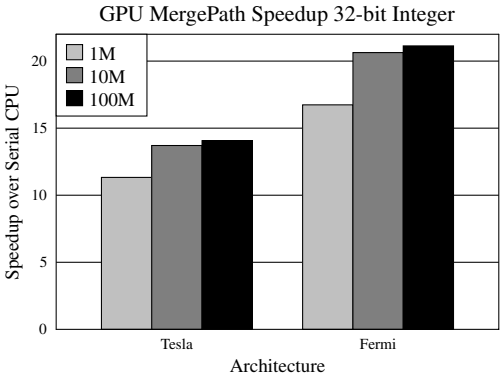


(a) 32-bit integer merging in OpenMP

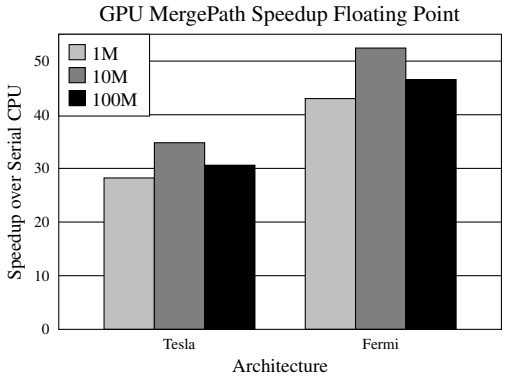


(b) Single-precision floating point merging in OpenMP

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

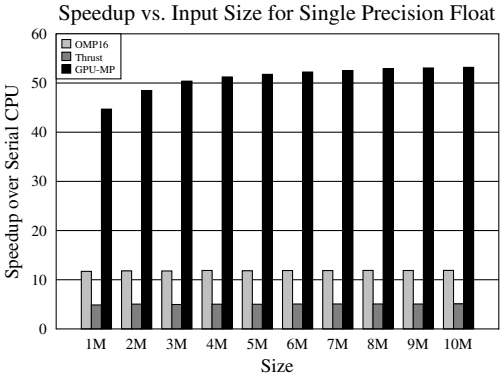


(a) 32-bit integer merging on GPU

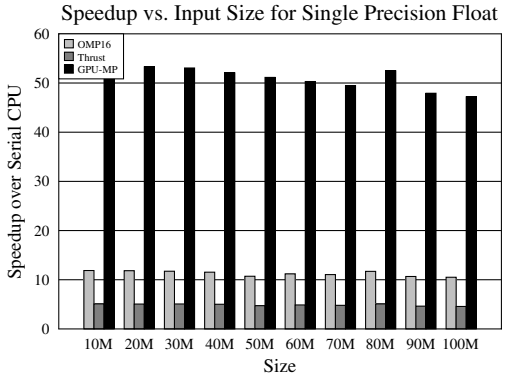


(b) Single-precision floating point merging on GPU

Figure 6: The speedup of the GPU implementations of Merge Path on the NVIDIA Tesla and Fermi architectures over the x86 serial merge.

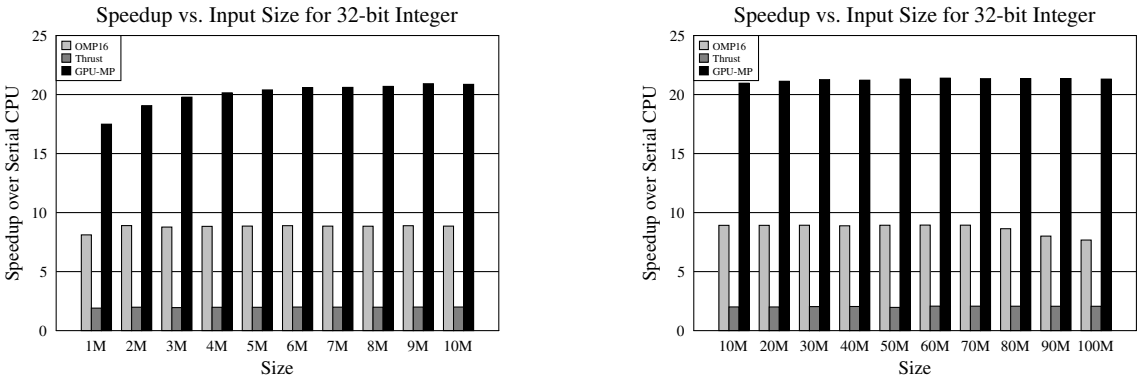


(a) 1M to 10M elements



(b) 10M to 100M elements

Figure 7: Speedup comparison of merging single-precision floating point numbers using Parallel Merge Path on 16 threads, Merge Path on Fermi (112 blocks of 128 threads), 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: Speedup comparison of merging 32-bit integers using Parallel Merge Path on 16 threads, GPU Merge Path on Fermi (112 blocks of 128 threads), and thrust::merge on Fermi. Size refers to the length of a single input array in elements.

2. The sizes of threads and thread blocks, and thus the total number of threads, are selected to fit the hardware. 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 times of the GPU Merge Path algorithm on the NVIDIA Tesla and Fermi GPU architectures with those of the Thrust parallel merge [5] on the same architectures and Merge Path on a x86 Intel Nehalem core system. The specifications of the GPUs used can be seen in Table 1, and the Intel multicore configuration can be seen in Table 2. For the x86 system we show results for both a sequential merge and for OpenMP implementation of Merge Path. Other than the Thrust library implementation, the authors could not find any other parallel merge implementations for the CUDA architecture, and therefore, there are no comparisons to other CUDA algorithms to be made. Thrust is a parallel primitives library that is included in the default installation of the CUDA SDK Toolkit as of version 4.0.

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 (1) the increased size of the local shared memory per SM from 16KB to 64KB, (2) the option of using all or some portion of the L1 local shared memory as a hardware-controlled cache, (3) increased total CUDA core count, (4) increased memory bandwidth, and (5) increased SM size from 8 cores to 32 cores. In our experiments, we use the full local 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 local shared memory.

An efficient sequential merge on x86 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 parallelism 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. 3. 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.

In Fig. 4, 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.

Larger local shared memories 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 local shared memory, performs the cross diagonal search on this smaller problem in local shared memory, then cooperatively writes back 512 elements to the global memory. In the Tesla implementation, only 32 threads per block

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

are used to merge 4 elements per window for a total of 128 merged-elements due to local shared memory size limitations and memory latencies. Speedup for the GPU implementation on both architectures is presented in Fig. 6 for sorted arrays of 1 million,10 million, and 100 millions elements.

As the GPU implementations are benchmarked in comparison to the x86 sequential implementation and to the x86 parallel 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 (dual socket with quad core processors). Speedups are depicted in Fig. 5. 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. 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. We show a 12x speedup for the hyper-threading configuration for ten million floating point elements.

The main focus of our work is to introduce a new algorithm that extends the Merge Path concept to GPUs. We now present the speedup results of 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.

3.1 Floating Point Merging

For the floating point tests, random floating point numbers are generated to fill the input arrays then sorted on the host CPU. The input arrays are 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 completes, 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. These kernels are timed.

For arrays of sizes 1 million, 10 million, 100 million we see significant speedups of 28X & 34X & 30X on the Tesla card and 43X & 53X & 46X on the Fermi card over the sequential x86 merge. This is depicted in Fig. 6. In Fig. 7, we directly compare the speedup of the fastest GPU (Fermi) implementation, the 16-thread OpenMP implementation, and the Thrust GPU merge implementation. We checked a variety of sizes. In Fig. 7 (a) there are speedups for merges of sizes 1 million elements to 10 million elements in increments of 1 millions elements. In Fig. 7 (b) there are speedups for merges of sizes 10 million elements to 100 million elements in increments of 10 millions elements. The results show that the GPU code ran nearly 5x faster than 16-threaded OpenMP and nearly 10x faster than Thrust merge for floating point operations. It is likely that the speedup of our algorithm over OpenMP is related to the difference in memory bandwidth of the two platforms.

3.2 Integer Merging

The integer merging speedup on the GPU is depicted in Fig. 6 for arrays of size 1 million, 10 million, 100 million. We see significant speedups of 11X & 13X & 14X on the Tesla card and 16 & 20X & 21X on the Fermi card over the sequenctial x86 merge. In Fig. 8, 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. 8 is 112 blocks, similar to the number of blocks used in the floating point sub-section. We used the same sizes of sorted arrays for integer as we did for floating point.

4. CONCLUSION

We show a novel algorithm for merging sorted arrays using the GPU. The results show significant speedup for both integer and floating point elements. While the speedups are different for the two types, it is worth noting that the 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 merging integers on the CPU is 2.5X faster than the execution time for floating point on the CPU. This explains the reduction in the the speedup of integer merging on the GPU in comparison with speedup of floating point merging.

The new GPU merging algorithm is atomic-free, parallel, scalable, and can be adapted to the different compute capabilities and architectures that are provided by NVIDIA. In our benchmarking, we show that the GPU algorithm outperforms a sequential merge by a factor of 20X-50X and outperforms an OpenMP implementation of Merge Path that uses 8 hyper-threaded cores by a factor of 2.5X-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, local 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

This work was supported in part by NSF Grant CNS-0708307 and the NVIDIA CUDA Center of Excellence at Georgia Tech.

We would like to acknowledge Saher Odeh and Prof. Yitzhak Birk from the Technion 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] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *Parallel and Distributed Processing Symposium, International*, may 2012.

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

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

[11] 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.