Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA

(Technical Report UMIACS-TR-2010-08)

Zheng Wei and Joseph JaJa
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies, University of Maryland
College Park, U. S. A
{zwei, joseph}@umiacs.umd.edu

ABSTRACT

We present a number of optimization techniques to compute prefix sums on linked lists and implement them on multithreaded GPUs using CUDA. Prefix computations on linked structures involve in general highly irregular fine grain memory accesses that are typical of many computations on linked lists, trees, and graphs. While the current generation of GPUs provides substantial computational power and extremely high bandwidth memory accesses, they may appear at first to be primarily geared toward streamed, highly data parallel computations. In this paper, we introduce an optimized multithreaded GPU algorithm for prefix computations through a randomization process that reduces the problem to a large number of fine-grain computations. We map these fine-grain computations onto multithreaded GPUs in such a way that the processing cost per element is shown to be close to the best possible. Our experimental results show scalability for list sizes ranging from 1M nodes to 256M nodes, and significantly improve on the recently published parallel implementations of list ranking, including implementations on the Cell Processor, the MTA-8, and the NVIDIA GeForce 200 series. They also compare favorably to the performance of the best known CUDA algorithm for the scan operation on the Tesla C1060.

KEYWORDS: PREFIX COMPUTATION, LIST RANKING, GPU, CUDA, PARALLEL COMPUTING

1. INTRODUCTION

It has been widely recognized that the *scan operation* on an array of elements plays a fundamental role in parallel processing [1,2,3]. This operation amounts to computing all the prefix sums of the elements stored in an array, and was recognized early on to be solvable by a *fast* and *work-optimal* data parallel algorithm. In this case, a fast parallel algorithm refers to *O(log n)* parallel time assuming an unlimited number of processing elements. By work optimal, we refer to the fact that the data parallel algorithm asymptotically employs the same total number of operations as the best sequential algorithm. A fast and work-efficient implementation of the scan operation on the NVIDIA GPU was recently reported in [4]. The computation of prefix sums on the elements contained in a linked list also plays an important role in parallel processing of irregular applications involving linked structures such as trees and graphs [5]. Given that the successor of each node of a linked list can appear anywhere in the memory, this computation can be dominated by fine-grain irregular memory accesses and as such it represents a challenging problem for parallel computing. It is worth noting that the *list ranking* problem is a special case in which all the values stored in the linked list are equal to the identity element. In such a case, the prefix sum of a node (called its *rank*) is equal to the distance of the node from the head of the list. In this paper, we will tackle the general prefix sums problem but sometimes refer to it as list ranking.

The development of parallel algorithms for list ranking has received significant attention in the literature dating back to the work of Wyllie [6] in which he introduced the pointer jumping technique for the PRAM model. More recently, the emergence of many internet applications that involve extremely large amounts of data with linked structures has rekindled interest in list ranking. Recent work on parallel algorithms for list ranking is reported in [7] and [8], which will later be compared to the approach developed in this paper.

Our main contributions can be summarized as follows.

- We develop a CUDA-tailored multithreaded implementation of a slight variant of the algorithm of Helman and JaJa [9] and provide an analysis that is used to determine optimal values for the number of blocks, number of threads, and number of sublists handled per thread.
- Experimental results on the NVIDIA Tesla C1060 show linear scalability for lists whose sizes range from 1M to 256M nodes, with the processing time per node shown to be very close to the best theoretical time that can be supported by the hardware.
- Our execution times are significantly faster than the recently published results for list ranking on multicore processors, and compare favorably to the best known CUDA implementation of the sequential scan operation on the same hardware.

2. MULTITHREADED GPUS USING CUDA

A surprising facet of the recent evolution of multicore processors is the continued appearance of extremely powerful GPUs that have much better performance to power ratio than the standard multicore CPUs, and that offer increasingly more flexible general purpose programming environments. Examples of such co-processors include the IBM Cell Broadband Engine (Cell BE), the NVIDIA GeForce 200 series, and the Intel Larrabee. These multicore processors provide flexible general-purpose programming environments with impressive peak performance. Since this paper is primarily concerned with optimization techniques for the NVIDIA CUDA programming model [10], we devote the rest of this section to briefly summarize that programming model and related features.

The basic architecture of the NVIDIA GeForce 200 series consists of a set of Streaming Multiprocessors (SMs), each of which containing eight Scalar Processors (SPs or cores) executing in a SIMD fashion, 16,384 registers, and a 16KB of shared memory. The 16KB shared memory is organized into 16 banks. Threads running on the same SM can share data and synchronize limited by the available resources on the SM. Each SM has small constant and texture caches. All the SMs have access to a very high bandwidth Global Memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However the latency to access the global memory is quite high and is around 400-600 cycles.

In our work, we have used the NVIDIA Tesla C1060 that has 30 SMs coupled to a 4GB global memory with a peak bandwidth of 102 GB/s. Fig. 1 and Fig. 2 illustrate the overall architecture of the Tesla C1060.

The characteristics of the memory organization of the Tesla C1060 are summarized in Table I.

The CUDA programming model envisions phases of computations running on a host CPU and a massively data parallel GPU acting as a co-processor. The GPU executes data parallel functions called *kernels* using thousands of threads. Each GPU phase is defined by a *grid* consisting of all the threads that execute some kernel function. Each grid consists of a number of *thread blocks* such that all the threads in a thread block are assigned to the same SM. Several thread blocks can be executed on the same SM, but this will limit the number of threads per thread block since they all have to compete for the resources (registers and shared memory) available on the SM. Programmers need to optimize the use of shared memory and registers among the thread blocks executing on the same SM.

Each SM schedules the execution of its threads into warps, each of which consists of 32 parallel threads. Half-warp (16 threads), either the first or second half of a warp, is introduced to match the 16 banks of shared memory. When all the warp's operands are available in the shared memory, the SM issues a single instruction for the 16 threads in a half-warp. The eight cores will be fully utilized as long as operands in the shared memory reside in different banks of the shared memory (or access the same location from a bank). If a warp stalls, the SM switches to another warp resident in the same SM.

Optimizing performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible and should be coalesced into multiple of contiguous 16-word lines); shared memory accesses (threads in a warp should access different banks); and partitioning of thread blocks among SMs; in addition to carefully designing highly data parallel implementations for all the kernels involved in the computation.

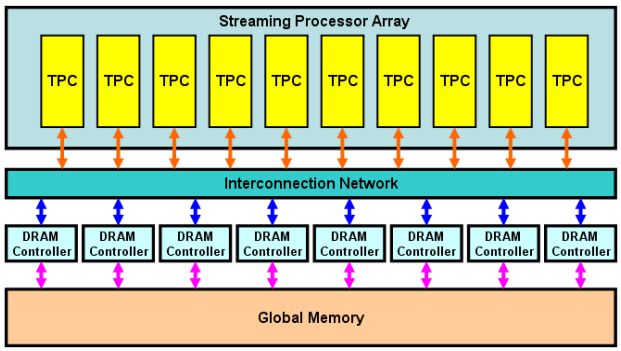

Figure 1. Architecture of the Tesla C1060

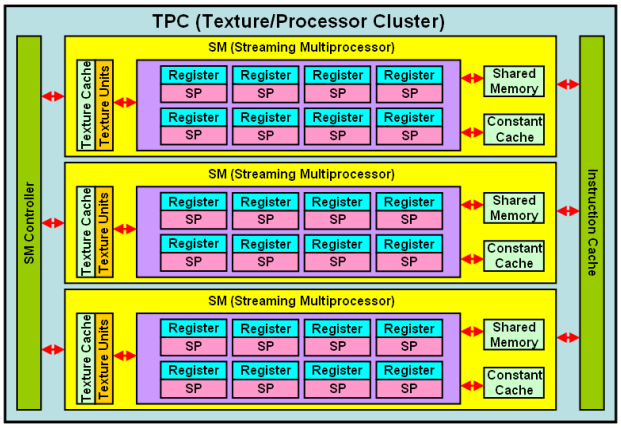

Figure 2. Architecture of the TPC in Tesla C1060

TABLE I. OVERALL MEMORY ORGANIZATION OF THE TESLA C1060

	Global Memory	**Shared Memory**	**Constant Cache**	**Texture Cache**
Size	4GB	16KB per SM	8KB per SM	6-8KB per SM
Accessibility	All threads	All threads in the same block	All threads	All threads
Other Properties	High latency	Low latency	Only readable by thread	Only readable by thread

3. MULTITHREADED LIST RANKING ALGORITHM AND OPTIMIZATION TECHNIQUES FOR CUDA

Let L be a singly linked list in which each node contains two fields, one containing a data element and the second containing a pointer to the successor. The last node on the list is distinguished by a negative value in the successor field. The computation of prefix sums on L amounts to updating the value in the data field of a node by the sum of the values stored in the data fields of all the predecessor nodes, including the node itself. In other words, it is identical to the scan operation when it is carried out on an array A such that $A[i]$ holds the value of the node of rank i. We assume that our list L is represented by an array X such that $X[i].prefix$ and $X[i].succ$ represent respectively the data and successor fields. Note that $X[i].succ$ could be any arbitrary index in the array X. In our case, we assume that the head of L is not known, and hence has to be identified by the algorithm before prefix computations can take place. The main reason for making this assumption is to explore the impact of the presence of significant caches (as in the Intel Clovertown processor) since the initial step that determines the head of the list will fill the cache with some of the input data thereby rendering the execution of later steps faster on such processors.

The list ranking problem admits of a simple and effective sequential algorithm involving two passes through the array X. The first pass identifies the head of the list, and the second pass traverses the list, starting from the head, and following the successor field while accumulating the prefix sums in the traversal order. This sequential algorithm performs extremely well in practice, especially if the processor contains a large enough cache, and hence it is somewhat of a challenge to develop a work-optimal, data parallel version that scales linearly.

3.1. Overview

Before introducing our approach, we quickly review the best known CUDA implementation of the scan operation on an array X [4], a considerably simpler problem. The array is partitioned into subarrays, each group of whose prefix sums is computed separately by a block of threads using a data parallel algorithm that is a slight variation of Blelloch's [3]. The prefix sums of the last element of each subarray are then written into another array Y, followed by applying the same scan procedure on Y. Each prefix sum of Y is then added to each element of the corresponding block of X. This strategy can be viewed as a divide-and-conquer strategy in which the conquer steps involve the use of data parallel algorithms. As is, this strategy will not be appropriate for the list ranking problem. In particular, each subarray resulting from the initial partition may contain many sublist fragments of the initial list, whose head nodes have to be identified before any useful work can be done. Such a process seems to require almost as much work as the initial list ranking problem.

Our approach for list ranking is also based on a divide-and-conquer strategy but the initial partitioning is randomized. We *randomly* select *s* nodes (called *splitters*) from *L* and use these nodes to decompose the list into *sublist fragments* (or just sublists), each of which begins with a random node and consists of the successor nodes until we reach a node whose successor is a splitter. As we will see later, the choice of *s* is critical for optimal performance; the larger *s* the better the load balance among the streaming processors but the more the overhead will be incurred in combining the partial results. We divide these fragments equally between CUDA grid blocks, and use highly data parallel algorithms to process the fragments within each block. This is followed by a list ranking algorithm applied on the list *L'* consisting of these *s* random nodes, where the successor of a random node *Z* is the first random node encountered upon the traversal of *L* starting from *Z*. The last merge step is similar to that of the scan operation and involves adding each prefix sum of *L'* to each element of the corresponding sublist fragment.

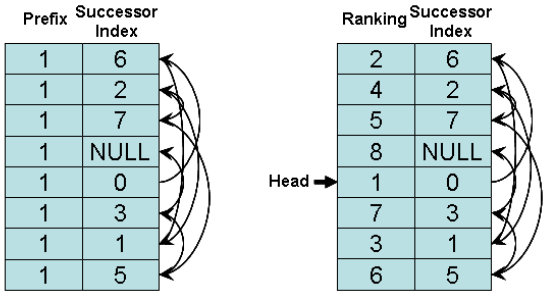

Figure 3. Input array and the corresponding output for list ranking

3.2. Detailed Description of Prefix Sums Algorithm

The prefix sums problem is formally defined as follows. We are given a singly linked list *L* of *n* elements stored in an array *X* such that *X[i]* contains two fields, *X[i].prefix* holding a data value and *X[i].succ* holding the array index of its successor. The successor of the last element of *L* contains a negative integer indicating the end of the list. We make the assumption that the index of the head of the list is not known. The prefix sum of the *i*th node (assuming not the head of the list) is defined by:

X[i].prefix = X[i].data $\otimes$ *X[pre].prefix*,

where *pre* is the index of the predecessor of *X[i]* and $\otimes$ can be any binary associative operator. For the head of the list, the corresponding prefix sum is equal to the value of the data stored there. Fig. 3 illustrates a simple example of a list and the corresponding prefix sums.

Our algorithm will follow the strategy outlined in the previous section using in particular many of the details developed in [9]. More specifically, our algorithm consists of the following five steps:

- Step 1: Compute the location of the head of the list.
- Step 2: Select *s* random locations of *X* to split the list into *s* random sublists, where each random location provides a pointer to the head of a sublist. As we will see, the value of *s* will be selected to guarantee load balancing with high probability.
- Step 3: Using the standard sequential algorithm, compute the prefix sums of each sublist separately. All the sublists can be handled in parallel.
- Step 4: Compute the prefix sums of the list consisting exclusively of the splitters, where the successor of a splitter is the next splitter encountered when traversing the initial list.
- Step 5: Update the prefix value of each element of the array *X* by using the prefix sum values computed in Step 4.

3.3. Implementation Details and Analysis

The list ranking algorithm in [9] was designed for symmetric multiprocessors with caches. In our heterogeneous CPU and GPU platform, we partition the work so that few specialized tasks that can substantially benefit from caches are allocated to the CPU. The remaining tasks are handled by thousands of threads on the GPU in such a way as to hide memory access latency and make effective use of the shared memory. The implementation details of each step are described next.

Step 1 can be implemented as follows. Since each of the indices between *0* and *n-1*, except for the index of the head node, must occur exactly once in the successor fields, the index of the head node can be computed by the formula:

$$HEAD = \frac{n(n-1)}{2} - SUM\_SUCC$$

where *SUM_SUCC* is the sum of the all indices in the successor fields, except for the negative index. Hence this step amounts to a sum operation, which can be performed by a data parallel CUDA algorithm built around a balanced binary tree [11]. The resulting algorithm is work optimal and makes effective use of global and shared memory accesses. The details can be found in [11].

Step 2 is implemented as follows. For every subarray of X of size n/s, we select a random location as a splitter, and record it in an array *Sublist_head*. Then the successor field of splitter i in the original list is first copied into array *Sublist_scratch[i]* and then changed into $-i$ in X to indicate the fact that it is now the head of sublist i. Hence Step 2 involves highly data parallel computations that are independent of each other.

Step 3, the most computationally demanding step of our algorithm, is implemented as follows. The s sublists are allocated equally among the CUDA blocks, which in turn are allocated equally among the threads of each block. Each thread will then compute the prefix sums of each of its sublists and copy the prefix value of the last element of sublist i into *Sublist_prefix[i]*. Note that the head location of sublist i can be found in *Sublist_head[i]*, and its successor index in *Sublist_scratch[i]*. The end of a sublist can be identified by the negative successor index, indicating the head of another sublist. Also, once a node is traversed, its successor is changed into $-j$ if it belongs to sublist j.

Using Lemma 1 of [9], the total number of nodes handled by a thread is about the same as any other thread with high probability if the number of sublists is at least $p \ln n$ and the number of processors p is $< \sqrt{\frac{n}{\ln n}}$, where n is the total number of nodes in the original list. We will later determine the best combination of the values of the number of blocks, the number of threads per block, and the number of sublists per thread, which will clearly satisfy these conditions.

Step 4 uses the standard sequential algorithm to carry out the prefix computations on the list of splitters. In our case, this step is executed on the CPU since the amount of data involved is quite small. As we will see, the amount of time taken by this step (including the time to move the list of splitters from the GPU global memory to the main memory and back into the GPU global memory) is less than 3% of the total execution time of the algorithm.

Step 5 updates the values of the prefix sums computed in Step 3 using the splitters prefix sums of Step 4. Hence we have to identify the sublist of *X[i]* for each i. This can easily be done by checking the value of *X[i].successor*, which was set to the negative index of the corresponding sublist. Therefore, all the elements of X can be updated as follows:

for all i between 1 and n do in parallel
    X[i].prefix= X[i].prefix $\otimes$ *Sublist_prefix[-X[i].successor]*

We perform this operation using coalesced memory accesses to the array X. We store *Sublist_prefix* into the constant memory of the Tesla C1060 and then each SM can use its constant memory cache to load *Sublist_prefix* instead of always fetching it from global memory.

D. Comparison with Recent Parallel List Ranking Algorithms

While both [7] and [8] use the algorithm of [9] as the basis of their list ranking algorithms for CUDA and the Cell Processor respectively, there are a significant number of differences between their implementations and ours. We focus here on the main differences with [8] and ours since they use the same CUDA programming model. The main differences are:

- The algorithm in [8] uses $n/\log n$ or $n/(2\log^2 n)$ as the number of splitters, which tends to generate too many sublists for very large n. As a result, handling the list of splitters and combining the results incur a very significant overhead. In our implementation we strike an optimal balance between the desirability of a large number of sublists (for fine-grain data parallel computations and load balancing) and the splitting/merging costs.
- The algorithm of [8] recursively computes the prefix sum of *Sublist_prefix*, which will incur a very significant overhead. We simply perform this step using a sequential algorithm on the CPU. In our case, the execution time of this step is at most 3% of the total time.
- It is not clear how the splitters in [8] are selected. Unless they are selected carefully, the longest sublist can be very long in which case no load balancing will be possible. In our algorithm, we select the splitters in such a way as to guarantee load balancing with high probability.
- Another difference is the fact that [8] assume that they already know the head of the list, which makes their problem slightly easier than ours.

4. EXPERIMENTAL RESULTS

Our prefix sums algorithm is tested on the NVIDIA Tesla C1060 graphic card as a co-processor to an Intel Xeon X5260 Processor. Each of the input arrays consists of 64-bit entries such that 32 bits are reserved for the data field and the remaining 32 bits are reserved for the successor field. We select addition as our associative $\otimes$ operation. As in [9], we consider three main types of input: (i) random, in which the successor is selected randomly from among the indices of the array; (ii) ordered, in which the successor node is the next consecutive node in the array; and (iii) stride of size d, in which the successor of each node is d indices away from the node, with wrapping around as necessary, and where d is some integer constant. We typically run each test hundreds of times, and report the average times over all these runs.

4.1. Performance as a Function of Various Parameters

We conduct extensive tests to determine the performance of our algorithm as a function of: (i) type of list (random, ordered, stride); (ii) number of random splitters (i.e. number of sublists). In the rest of this section, we give a summary of the results of these tests.

4.1.1. Performance as a Function of the types of lists

Typical execution times of our algorithm on ordered, random, and stride (with different stride values) lists with 64M (1M = 2^{20}) nodes are shown in Fig. 4. Tests run using different list sizes show a very similar pattern. Parameters (number of sublists, number of blocks, number of threads per block, and number of sublists per thread) are selected to yield the best performance for each type. The values of these parameters turn out to be the same for all list types. The results show a slightly faster performance on ordered lists, and show performance on stride lists which ranges from that of ordered lists to that of random lists depending on the size of the stride. However, overall the difference in execution times is at most 10%.

Given the overall memory architecture of the Tesla, these results are not surprising. Note that since parallel accesses to contiguous locations can be coalesced, we see a slightly improved performance on ordered lists, or stride lists with small strides. However memory coalescing is limited once parallel threads process different sublists that are relatively far apart. An important observation is that when the stride value is large enough (> 100), the performance on stride lists is almost identical or worse to that on random lists. Given this fact, we use stride lists with different stride values (typically stride value = 1,001) to report on the performance of our algorithm since it is much easier to generate such lists.

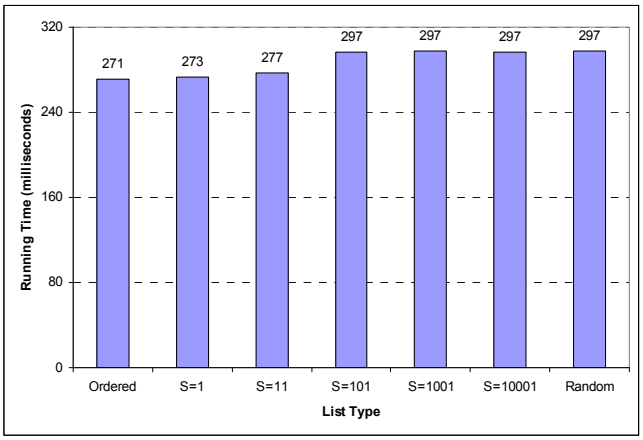

Figure 4. Performance of our algorithm on different types of lists with 64M nodes

4.1.2. Performance as a Function of the number of sublists

We focus our attention here in determining the best value of *s*, the number of sublists. This issue is somewhat intertwined with the choices of other parameters such as number of blocks, number of threads per block, and number of sublists per thread.

Since the head of each sublist is randomly selected, the length of each sublist may vary over a large range. To achieve the best performance, we must try to balance the work load (number of nodes processed) among all the SMs and the SPs, which is in particular critical for the most computationally demanding Step 3 of our algorithm. In fact, the running time of Step 3 is dominated by the execution time of the SM that has to process the maximum number of nodes.

For a fixed value of *s*, it is clear from the randomization performed in Step 2 that the more sublists are assigned to each thread, the more load balanced the work of the threads will be. This is intuitively clear and also follows from the proof of the main lemma in [9]. Hence this implies that a smaller number of threads will yield better performance. We can make the same argument regarding the number of blocks. For our Tesla processor, the number of blocks has to be a multiple of 30 (including 30 itself) and the number of threads per block has to be a multiple of 64. It turns that, for a fixed size list, the best performance is achieved when the number of blocks is equal to 30 and the number of threads per block is set equal to 64. Fixing these two parameters, we now take a close look at the execution time of Step 3 and overall execution time of the algorithm as a function of the number of sublists for a list of size 64M.

As expected, the execution time of Step 3 decreases as the number s of sublists increases. However the overhead to create these lists (Step 2) and to combine the partial results (Step 4) will eliminate any gain beyond the valued of *s*=61,440, resulting from the combination of 30 blocks, 64 threads per block, and 32 sublists per thread. It turns out that these values are consistent for all lists of sizes larger than 16M (up to 256M in our tests). Also, note that in the same graph we show that the maximum number of nodes handled by any SM, which decreases as we increase the number of splitters.

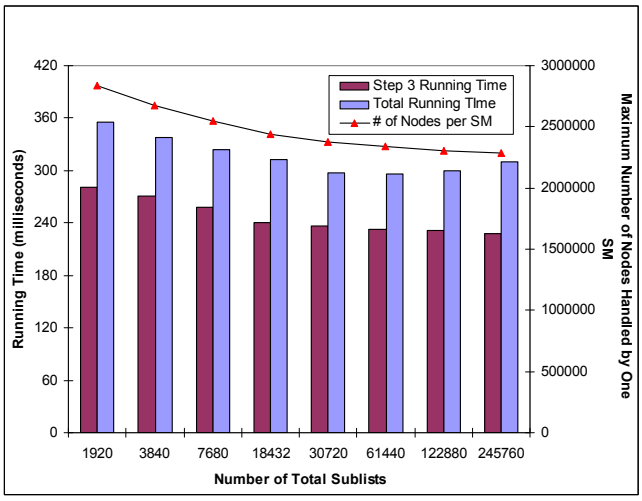

Figure 5. Performance of Step 3 and overall algorithm as a function of *s* on 64M stride list (stride = 1001). Upper curve represents the maximum load on an SM as a function of *s*.

4.1.3. Performance as a Function of the number of blocks

In this section, we report on the performance of our algorithm as we vary the number of blocks from 1 to 120. The NVIDIA scheduler allocates each block to an SM, and may allocate up to eight blocks to a single SM as necessary. Fig. 6 shows that the performance of our algorithm is almost linear in the number of blocks whenever the number is less than or equal to 30. In our tests, we have used a stride list of 64M nodes with stride value 1001, 64 threads per block, and 61,440 sublists. This combination yields the best performance for any number of blocks. The performance stays about the same as we increase the number of blocks from 30 to 120, indicating that load balancing and memory accesses do not improve as we increase the number of blocks beyond 30.

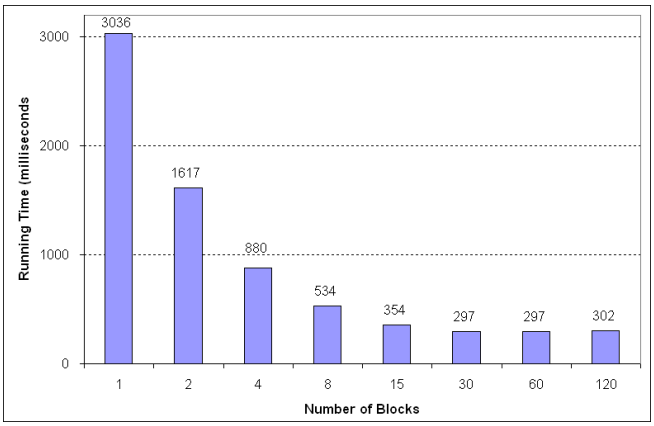

Figure 6. Performance of our algorithm on a 64M stride list (stride = 1001) as a function of the number of blocks.

4.2. Migration of the computations in Step 4 to CPU

In Step 4, the sequential prefix sums algorithm is executed on the splitters, a process that can be carried out on the CPU or the GPU. If this task is migrated to the CPU, we have to pay the additional cost of moving the data from the GPU global memory to main memory and then copying the results back into the GPU to complete the execution of the algorithm. A reason to do this is that all the splitters can fit into CPU cache, which can speed up the computation. In our tests on 64M lists with 61,440 splitters, the GPU sequential algorithm takes 56.72 milliseconds to finish this task, which is about 16.57% of the total running time. On the other hand, this step only takes 2.076 milliseconds if migrated to the CPU: 0.346 milliseconds to copy these splitters from the GPU memory to main memory, 1.365 milliseconds to perform the computation, and 0.365 milliseconds to copy the results back to the GPU. This amounts to 0.70% of the total running time and the percentage is even lower for larger lists.

4.3. *Overall Performance of our Algorithm and its Scalability*

We now report on the overall performance of our algorithm as a function of the list size using stride lists whose sizes range from 1M to 256M. Table II shows the detailed execution times of each of the steps for different list sizes. We have used the number of blocks to be 30 and the number of threads per block to be 64 based on the discussion above. The number of sublists per thread increases linearly with list size until it reaches 32, after which the running time spent on dividing and conquering increases faster than the time saved in fine tuning the loads among the threads. The overall running time scales linearly with the size of the list as illustrated by the graph in Fig. 7.

TABLE II. EXECUTION TIMES IN SECONDS OF THE DIFFERENT STEPS OF OUR ALGORITHM AS A FUNCTION OF THE LIST SIZE. WE USE STRIDE LISTS (STRIDE VALUE = 1001)

List Size (1M=2^{10})	Sublists per Thread	Step 1 (secs)	Step 2 (secs)	Step 3 (secs)	Step 4 (secs)	Step 5 (secs)	Total (secs)
1M	30*64*2	0.000221	0.000350	0.003877	0.000118	0.000824	0.005592
2M	30*64*4	0.000387	0.000744	0.007126	0.000468	0.001740	0.010718
4M	30*64*8	0.000669	0.000639	0.014794	0.000476	0.003473	0.020342
8M	30*64*16	0.001261	0.001140	0.028548	0.000979	0.006164	0.038470
16M	30*64*32	0.002451	0.001944	0.057316	0.001723	0.011974	0.075690
32M	30*64*32	0.004889	0.002139	0.115246	0.002380	0.024180	0.149186
64M	30*64*32	0.011538	0.003580	0.225137	0.002076	0.050643	0.296722
128M	30*64*32	0.019263	0.002129	0.451424	0.002006	0.099795	0.574955
256M	30*64*32	0.038433	0.002682	0.935964	0.002001	0.201184	1.180574

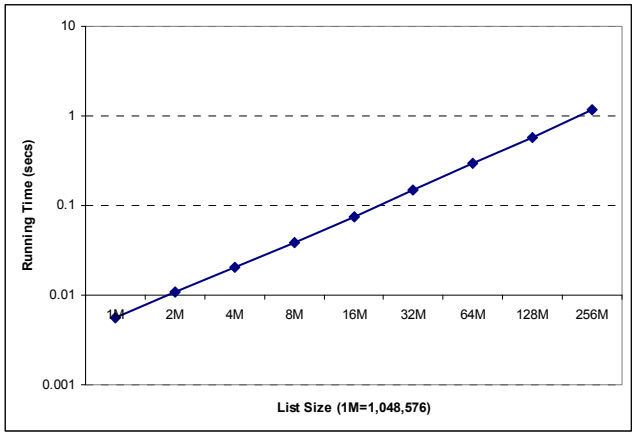

Figure 7. Scalability to list sizes on Stride lists (stride = 1001)

4.4. *How much Can our Algorithm be Improved?*

We now take a close look at the processing time per node achieved by our algorithm as a function of the list size ranging from 1M to 256M. Based on our extensive tests, the processing cost per node ranges between 4.28ns to 5.33ns, as long as the stride value is not extremely large. However, and surprisingly, the processing cost per node seems to increase significantly when the stride value is extremely large. To better understand this unexpected phenomenon and to determine whether our cost per node can theoretically be improved, we develop an optimized CUDA program whose main goal is to only access the device global memory at a very fine granularity, and compare its execution time per element to that of our prefix sums algorithm. It turns out that the performance of our algorithm is very close to the optimized global memory testing algorithm, both for extremely large strides and for moderate/small strides. We provide the details next.

Let Y be an array such that each entry consists of 64 bits. We develop a trivial CUDA algorithm such that each of its threads reads a location of Y (dependent on the thread ID), and then successively reads N locations which are at a certain stride d from the initial location. Our goal is to develop such a CUDA algorithm that results in the best possible performance per single access. Let B be the number of blocks and Th be the number of threads of our program. We conduct tests based on the following 224 possible combinations of the values of B, Th, and N:

B = { 8, 15, 16, 30, 32, 60, 64, 128 };
Th = { 8, 16, 32, 64, 128, 256, 512};

N = { 1024, 2048, 4096, 8192}.

The ranges above seem to cover all reasonable values to achieve the best possible access cost per element. After exploring all the combinations, the values achieving the best access time per element turn out to be: B = 16, Th = 64 and N = 4096. Some other combinations came close but not as good as this combination. The typical time per access is 4.044067 ns for many of the tested sizes for the array Y (varied from 1M to 256M entries – that is, actual size of array Y varies from 8MB to 2GB).

We now compare the performance of the global memory testing algorithm with that of our prefix sums algorithm focusing on stride lists, starting with the case of stride value d =1001. A summary of the results is shown in Table III, which is also illustrated in the graph shown in Fig. 8. These results clearly demonstrate that the performance of our list ranking program is very close to the performance achieved by the optimized global memory testing program.

TABLE III. TIME IN NANOSECONDS PER ELEMENT COMPARISON BETWEEN LIST RANKING AND GLOBAL MEMORY TESTING PROGRAM (STRIDE = 1001)

List Size (1M = 1,048,576)	1M	2M	4M	8M	16M	32M	64M	128M	256M
File Size (MB)	8	16	32	64	128	256	512	1024	2048
Test Program-Stride=1001 (per 64-bit integer)	3.70	3.84	3.93	3.97	4.00	4.04	4.06	4.13	4.19
List Ranking-Stride=1001 (per 64-bit node)	5.33	5.11	4.85	4.59	4.51	4.45	4.42	4.28	4.40

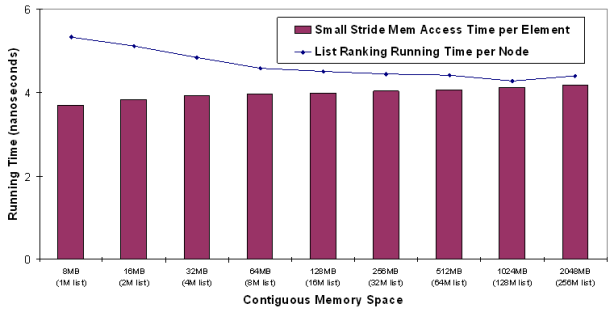

Figure 8. Time per element comparison between list ranking and global memory testing program (Stride = 1001) on the Tesla C1060

We now consider the case of extremely large stride values which are close to the size of lists. In this case, the performance degrades substantially when the stride value is 64M or larger (and hence list sizes are larger than 64M nodes) both by the global memory testing program as well as our prefix sums algorithm. However the performance per element still matches for both algorithms. Table IV and Fig. 9 illustrate the performance of both algorithms for extremely large stride values.

TABLE IV. PERFORMANCE COMPARISON OF COST PER ELEMENT IN NANOSECONDS BETWEEN LIST RANKING AND GLOBAL MEMORY TESTING PROGRAM USING EXTREMELY LARGE STRIDES

List Size (1M = 1,048,576)	1M	2M	4M	8M	16M	32M	64M	128M	256M
File Size (MB)	8	16	32	64	128	256	512	1024	2048
Test Program-Random (per 64-bit integer)	3.74	3.86	3.95	3.99	4.03	4.06	4.07	8.70	13.40
List Ranking (per 64-bit node)	5.33	5.11	4.85	4.59	4.51	4.45	4.42	9.33	14.05

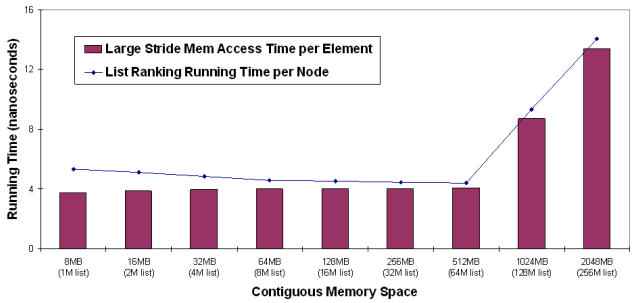

Figure 9. Time per element comparison between list ranking and global memory testing program with extremely large strides

4.5. Performance Comparison on Different Machines

In this section, we compare the performance of our prefix sums algorithm on the Tesla C1060 and the Intel 8-core Clovertown, as well as, to the performance of other recently reported list ranking algorithms on the Cell Processor, the MTA-8 (eight 220MHz Cray MTA-2 processors) [7], and the NVIDIA GTX280 [8]. Our Intel Clovertown consists of four duo-cores, each with 4MB of L2 cache. The overall Clovertown architecture is illustrated in Fig. 10.

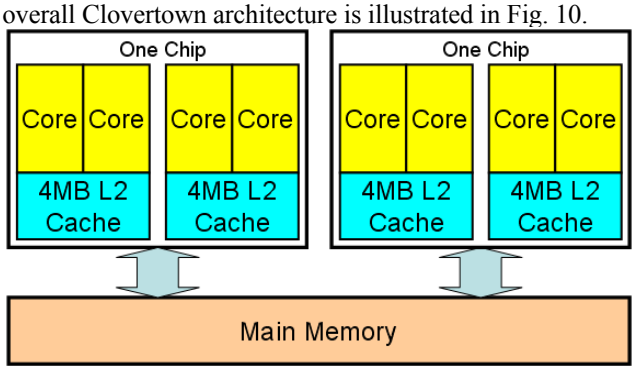

Figure 10. An overview of the architecture of the Clovertown

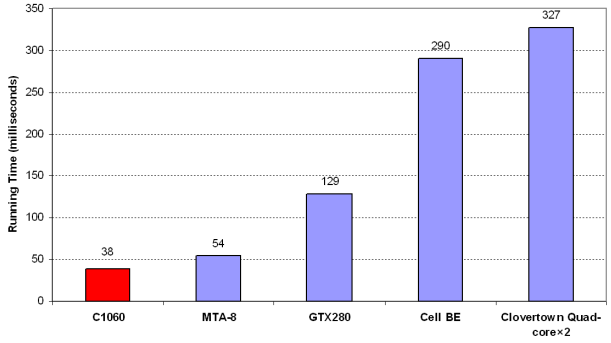

Figure 11. Comparison of different platforms on Random List with 8M nodes

Fig. 11 provides a summary of the overall performance of our prefix sums algorithm on the Tesla and Clovertown, and the best performance numbers reported on other machines for the list ranking problem on random list with 8M nodes. It is clear that our algorithm achieves the best raw performance even when compared to the Cray MTA-8 machine. The high performance of our algorithm can be attributed to the following facts:

- Our implementation uses thousands of active threads to make effective use of the hundreds of processors available and to hide memory access latency;
- Cache sensitive computations are migrated to the CPU while the Cell has a small local cache and the MTA-8 has no local cache;
- The tradeoff between balanced workload for each processor and a small overhead of merging parallel results is well addressed and as a result the power of the underlying GPU hardware is fully utilized;
- Shared memory with one clock cycle access time among eight SPs from the same SM is used whenever applicable to reduce the inter-processor synchronization and communication overhead of parallelism; on the other hand, processors on either the Cell or the MTA-8 are interconnected by networks with relatively high latency;
- Global memory accesses are coalesced to exploit the high bandwidth of the GPU whenever possible.

We now compare the performance of our algorithm to that of the scan operation on the Tesla C1060, as well as to that of the Clovertown on ordered lists. Clearly we expect the Clovertown performance to improve significantly on ordered lists because of the L2 cache. On the other hand, the scan operation is specifically designed for sequential lists, and hence optimized coalescing of global memory accesses and optimized access to the shared memory (that is, avoiding bank conflicts) are reported in the CUDA implementation of [4]. A summary of the results is illustrated in Fig. 12 for the case of 64M nodes. Both the scan algorithm and our algorithm are run on the Tesla C1060, and the performance ratio 7-8 of our algorithm compared to the scan operation seems to hold independent of the list size.

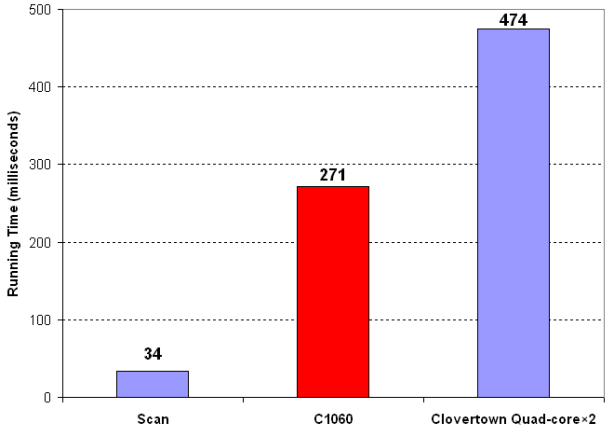

Figure 12. Performance of the scan operation and our list ranking algorithm applied to 64M ordered list on the Tesla C1060. The rightmost bar represents the performance of our algorithm on the Clovertown using the same ordered list.

5. CONCLUSION

In this paper, we presented an optimized CUDA algorithm for performing prefix sums on linked lists. Such computation amounts to highly irregular, fine-grain global memory accesses, and hence is usually considered to be unsuitable for the stream based, data parallel CUDA programming model. The algorithm creates randomized sublists that are handled by parallel threads in such a way that the merging operation is organized to exploit the CUDA architecture. We have conducted extensive tests of our algorithm on the Tesla C1060 using different types of linked lists of sizes ranging from 1M nodes to 256M nodes. The results show scalable performance with the cost of processing a node close to the best possible for the Tesla processor. A byproduct of our tests is the discovery of significant memory performance degradation whenever the consecutive locations accessed by a thread become far apart.

ACKNOWLEDGMENT

We would like to thank Dave Luebke and NVIDIA for providing the Tesla processors used in this research and Mark Harris from NVIDIA for providing help in CUDA programming. This research was partially supported through an NSF Research Infrastructure award, grant number CNS 0403313.

REFERENCES

[1] G. E. Blelloch, Vector Models for Data-Parallel Computing, MIT Press, Cambridge, London, 1990.

[2] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, Scan primitives for GPU computing, Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2007, pp. 97-106.

[3] G. E. Blelloch, Prefix sums and their applications, Chapter 1 in Synthesis of Parallel Algorithms by J. H. Reif, Morgan Kaufmann Publishers Inc., San Mateo, California, 1993, pp. 35-60.

[4] M. Harris, Parallel Prefix Sum (Scan) with CUDA, NVIDIA Corporation, 2008.

[5] J. JaJa, And Introduction to Parallel Algorithms, Addison-Wesley Publishing Company, New York, 1992.

[6] J. C. Wyllie, The Complexity of Parallel Computations, PhD Thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

[7] D. A. Bader, V. Agarwal, K. Madduri, On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking, IEEE International Parallel and Distributed Processing Symposium, 2007.

[8] M. S. Rehman, K. Kothapalli, P. J. Narayanan, Fast and Scalable List Ranking on the GPU, 23rd International Conference on Supercomputing (ICS), New York, USA, 2009.

[9] D.R. Helman, J. JaJa, Prefix Computations on symmetric multiprocessors, Journal of Parallel and Distributed Computing, 2001, 61(2): pp. 265-278.

[10] NVIDIA Corporation, NVIDIA CUDA Programming Guide Version 2.2.1, 2009.

[11] M. Harris, Optimizing Parallel Reduction in CUDA, available at http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf. Access date: 01/27/2010.