Fast and Scalable List Ranking on the GPU

M. Suhail Rehman, Kishore Kothapalli*, P. J. Narayanan†
International Institute of Information Technology
Hyderabad, India
{rehman@research., kkishore@, pjn@}iiit.ac.in

ABSTRACT

General purpose programming on the graphics processing units (GPGPU) has received a lot of attention in the parallel computing community as it promises to offer the highest performance per dollar. The GPUs have been used extensively on regular problems that can be easily parallelized. In this paper, we describe two implementations of List Ranking, a traditional irregular algorithm that is difficult to parallelize on such massively multi-threaded hardware. We first present an implementation of Wyllie's algorithm based on pointer jumping. This technique does not scale well to large lists due to the suboptimal work done. We then present a GPU-optimized, Recursive Helman-JáJá (RHJ) algorithm. Our RHJ implementation can rank a random list of 32 million elements in about a second and achieves a speedup of about 8-9 over a CPU implementation as well as a speedup of 3-4 over the best reported implementation on the Cell Broadband engine. We also discuss the practical issues relating to the implementation of irregular algorithms on massively multi-threaded architectures like that of the GPU. Regular or coalesced memory accesses pattern and balanced load are critical to achieve good performance on the GPU.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming, Parallel Programming*

General Terms

Algorithms, Design, Performance

Keywords

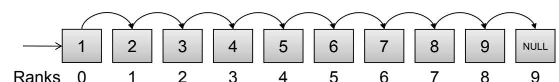
GPGPU, list ranking, many-core, parallel algorithm, irregular algorithm

*Center for Security, Theory, and Algorithmic Research
†Center for Visual Information Technology

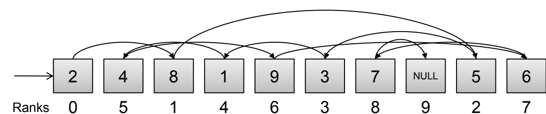
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...$5.00.

1. INTRODUCTION

Commodity graphics hardware has evolved into highly parallel and fully programmable architectures. The latest GPUs like NVIDIA's GTX280 GPU and AMD's HD4870 contain several hundreds of processing cores and provides a theoretical performance of 1 TFLOP. While the traditional GPGPU approach used the graphics API to perform general purpose computations, programming models and drivers to exploit the computing power using a general purpose stream programming interface has made it easier to develop applications on the GPUs. The Compute Unified Device Architecture (CUDA) exposes an alternate programming model on NVIDIA GPUs that is close to the traditional PRAM model [16]. The recent adoption of the OpenCL[15] as an open, multi-vendor, standard API for high-performance computing has the potential to bring these devices to the mainstream of computing due to portability across GPUs, multi-core CPUs, and other accelerators.



(a) Ordered list.



(b) Random list.

Figure 1: Linked lists and their corresponding ranks.

The GPU architecture fits the data parallel computing model best, with a single processing kernel applied to a large data grid. The cores of the GPU execute in a Single Instruction, Multiple Data (SIMD) mode at the lowest level. Many data parallel applications have been developed on the GPU in the recent years [6], including FFT [9] and other scientific applications [10]. Primitives that are useful in building larger data parallel applications have also been developed on the GPUs. These include parallel prefix sum (scan), reduction, and sorting [19]. Regular memory access and high arithmetic intensity are key to extracting peak performance on the GPUs. Problems that require irregular or random memory accesses or sequential compute dependencies are not ideally suited to the GPU. The list ranking problem [7,

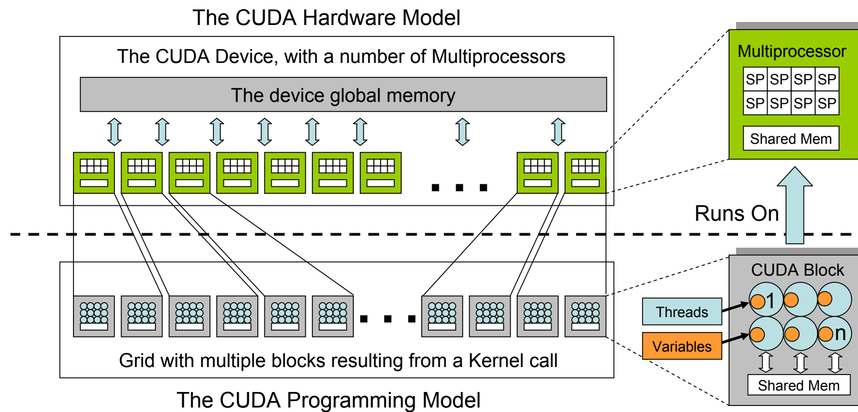


Figure 2: The CUDA Computation Model [16].

17, 13] is a typical problem that has irregular memory access patterns and sequential dependencies.

List ranking and other computations on linked lists are fundamental to the handling of general structures such as graphs on parallel machines. The importance of list ranking to parallel computations is identified by Wyllie [21]. While it is easy to solve in the sequential setting, algorithms in the parallel setting are quite non-trivial and differ significantly from known sequential algorithms. The range of techniques deployed to arrive at efficient parallel list ranking algorithms include independent sets, ruling sets, and deterministic symmetry breaking among others.

Wyllie [21] gave a simple algorithm that can be used for list ranking. However, the algorithm is not work-optimal [13], which means that the algorithm performs more than $O(n)$ operations for a list of n elements. The first optimal algorithm is given by Cole and Vishkin [7]. Anderson and Miller [1] proposed another optimal algorithm using ideas from independent sets. Their algorithm, while being deterministic, is however not easy to implement. A simpler randomized algorithm is proposed by Hellman and JáJá [12] using ideas of sparse ruling sets from Reid-Miller [17]. The algorithm of Hellman and JáJá [12] is worst-case work-optimal for a small number of processors. Another implementation for distributed systems using a divide-and-conquer approach is reported by Sibeyn [20].

List ranking is a basic step of algorithms such as Euler tour [13], load balancing, tree contraction and expression evaluation [5], connected components [4], planar graph embedding etc. [14]. The non-contiguous structure of the list and irregular access of shared data by concurrent threads/processes make list ranking tricky to parallelize. Unlike the prefix sum or scan operation, there is no obvious way to divide a random list into even, disjoint, continuous sublists without first computing the rank of each node. Concurrent tasks may also visit the same node by different paths, requiring synchronization to ensure correctness.

We explore the list ranking problem on the GPU using the CUDA computation model in this paper. We first present an implementation of the pointer jumping algorithm on the GPU. The requirements of atomic operations for concurrent updates result in the GPU algorithm being faster only on lists of half a million or more elements. We then present a recursive formulation of the Helman-JáJá algorithm which outperforms the CPU algorithm on lists of 32K or more elements. We obtain significant speedups on larger lists compared to the CPU implementation and the implementation on the Cell BE[3] and SMP[2] machines. We also analyze the factors that contribute to the performance on the GPUs. This analysis is relevant to implementing other irregular algorithms on the GPUs. Our main contributions are summarized as follows:

1. We present an implementation of Wyllie's Algorithm which is suboptimal due to its memory operations on the GPU and workload and does not scale well to large lists.

2. We present a fast and scalable recursive implementation of Hellman and Jáá's list ranking algorithm on massively multi-threaded architectures, which outperforms all previous reported implementations on similar architectures. On a random list of 8 million nodes, our implementation is about 3-4 times over the best reported implementation on the Cell Broadband Engine.

3. We conduct an empirical study on the effects of load-balancing and coalescing of memory accesses for irregular algorithms on the GPU. This study can be of independent interest when one implements such algorithms on the GPU.

4. We conclude that the major issues that dictate performance while implementing similar algorithms on the GPU include: exploitation of massive parallelism, efficient use of global memory, and load balancing among threads.

2. GPU COMPUTATION MODEL

The GPU is a massively multi-threaded architecture containing hundreds of processing elements or *cores*. Each core comes with a four stage pipeline. Eight cores are grouped in SIMD fashion into a *symmetric multiprocessor (SM)*, hence all cores in an SM execute the same instruction. The GTX280 has 30 of these SMs, which makes for a total of 240 processing cores.

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and multiple blocks are grouped into *grids*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads. An SM executes one warp at a time. CUDA has zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp. For this purpose, NVIDIA recommends 1024 threads (across multiple blocks) be assigned to an SM to keep it fully *occupied*. The various resources and configuration limits of the GTX 280 GPU under CUDA is shown in Table 1.

The GPU also has different types of memory at each level (Figure 2(a)). A set of 32-bit registers is evenly divided among the threads in each SM. Scratchpad memory of 16 KB, known as *shared memory*, is present at every SM and can act as a user-managed cache. This is shared among all the blocks scheduled on an SM. The GTX 280 also comes with 1 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but incurs hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp. The general, read-write access of the global memory is not cached. Thus, locality of memory access over time by a thread provides no advantages. Simultaneous memory accesses by multiple adjacent threads, however, are *coalesced* into a single transaction if they are adjacent.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Each kernel executes a grid. Prior to launching a kernel, all the data required for the computation must be transferred from the host (CPU) memory to the GPU (global) memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data (Figure 2(b)). Barrier synchronization for all threads in a block can be defined by the user in the kernel code. Apart from this, all threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be preformed across separate kernel launches.

The irregularity of the list ranking problem's memory accesses make it very difficult to apply conventional GPU optimizations provided by CUDA. Since the memory access pattern is not known in advance and there is no significant data reuse in each phase of the algorithm, we cannot take advantage of the faster shared memory.

Resource or Configuration Parameter	Limit
No of Cores	240
Cores per SM	8
Threads per SM	1,024 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	16,384 registers
Active Warps per SM	32 warps

Table 1: Constraints of GTX 280 on CUDA.

3. LIST RANKING ON THE GPU

The traditional approach to implementing a data-parallel algorithm in CUDA is to launch a CUDA kernel for each task. It is advisable to stick to this model as far as possible, even if it means doing a bit of extra work in each thread, as it is a massively parallel architecture.

3.1 Wyllie's Algorithm

Wyllie's algorithm[21] involves repeated pointer jumping. The successor pointer of each element in the list is repeatedly updated so that it jumps over its successor until we reach the end of the list. As each processor traverses and contracts the successor of a list, the ranks are updated for each jump. Once all the processors have contracted all the way to the end of the list, the algorithm ends. Wyllie's algorithm also provides an opportunity to present an implementation of a pointer-jumping implementation on the GPU which will be useful to various other parallel algorithms that use this as a primitive [13].

Algorithm 1 Wyllie's Algorithm

Input: An array S, containing successors of n nodes and array R with ranks initialized to 1
Output: Array R with ranks of each element of the list with respect to the head of the list

```
1: for each element in S do in parallel
2:     while S[i] and S[S[i]] are not the end of list do
3:         R[i] = R[i] + R[S[i]]
4:         S[i] = S[S[i]]
5:     end while
6: end for
```

Given a list of size n, the implementation of this algorithm (Algorithm 1) is to assign a process/thread per element of the list. Each thread performs $O(\log n)$ steps, thereby making the total work complexity $O(n \log n)$. By the end of the algorithm, all ranks would have been updated correctly. It is important to note that the rank, successor and finished bit writing step (step 10) in the algorithm must be performed concurrently for each thread. If a process updates either the rank or successor value of an element in between the update operation of another process, the algorithm will fail.

CUDA does not have any explicit user management of threads or memory synchronization locks. Hence, for this implementation in CUDA, we need to pack two single precision words (elements R_i and S_i) into a double word and perform a 64 bit write operation(Figure 3). It must be noted that 64-bit operations are more expensive on the GPU. Hence for lists of size $\leq 2^{16}$, the two 16-bit integers (representing $R[i]$ and $S[i]$) can be packed into a 32-bit integer and use a 32-bit write operation. This can result in time savings of up to 40% on such lists.

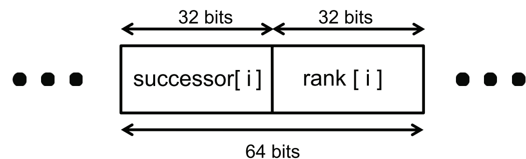


Figure 3: Packing two 32-bit values into a double word.

This algorithm is further modified for the GPU to load the various data elements from the global memory only when

they are needed, as global memory read operations are expensive. The CUDA Kernel implementation of this algorithm is in Listing 1. The current element, `LIST[index]`, is first read in `node`. The successor node is also loaded in `next`. Using bitwise operations, the rank and successor are read from these variables and updated as required. Finally after packing the new values in `temp`, the new node is written back to `LIST[index]`. We synchronize all the threads in a block using `__syncthreads` to force all the eligible threads to read/write in a *coalesced* manner [16]. These operations are looped until all the nodes have set their successors to -1 (which denotes the end of the list).

Listing 1: Wyllie's Algorithm implemented as a GPU Kernel in CUDA

```
__global__ void ListRank
(long long int *LIST, int size)
{
  int block=(blockIdx.y*gridDim.x)+blockIdx.x;
  int index=block*blockDim.x+threadIdx.x;

  if(index<size)
  {
    while (1)
    {
      long long int node = LIST[index];
      if (node>>32 == -1) return;
      __syncthreads();

      int mask=0xFFFFFFFF;
      long long int temp=0;
      long long int next = LIST[node>>32];

      if (next>>32 == -1) return;

      temp = (int) node &  mask;
      temp += (int) next & mask;
      temp += (next>>32)<<32;

      __syncthreads();

       LIST[index]=temp;
    }
}
```

3.2 Helman and JáJá Algorithm and its Recursive Variant

Helman and JáJá's algorithm was originally designed for symmetric multiprocessors (SMP) [12]. The parallel algorithm for a machine with p processors is as follows:

Figure 4 illustrates the working of the algorithm on a small list. This algorithm (Algorithm 2) splits a random list into s sublists in the splitting phase (Figure 4(a)). The rank of each node with respect to the "head" of each sublist is computed by traversing the successor array until we reach another sublist (local ranking phase - Figure 4(b)). This is done in parallel for s sublists. The length of each sublist is used as the prefix value for the next sublist. A new list containing the prefixes is ranked sequentially (global ranking phase - Figure 4(c)) and then subsequently matched and added to each local rank in the final phase (recombination phase). Helman and JáJá proved that for a small constant, when p is small, the worst case run time is $O\left(\log n + \frac{n}{p}\right)$ with $O(n)$ work [12].

Algorithm 2 Helman and JáJá List Ranking Algorithm

Input: An array L, containing input list. Each element of L has two fields *rank* and *successor*, and $n = |L|$

Output: Array R with ranks of each element of the list with respect to the head of the list

1: Partition L into $\frac{n}{p}$ sublists by choosing a splitter at regular indices separated by p.
2: Processor p_i traverses each sublist, computing the local (with respect to the start of the sublist) ranks of each element and store in $R[i]$.
3: Rank the array of sublists S sequentially on processor p_0
4: Use each processor to add $\frac{n}{p}$ elements with their corresponding splitter prefix in S and store the final rank in $R[i]$

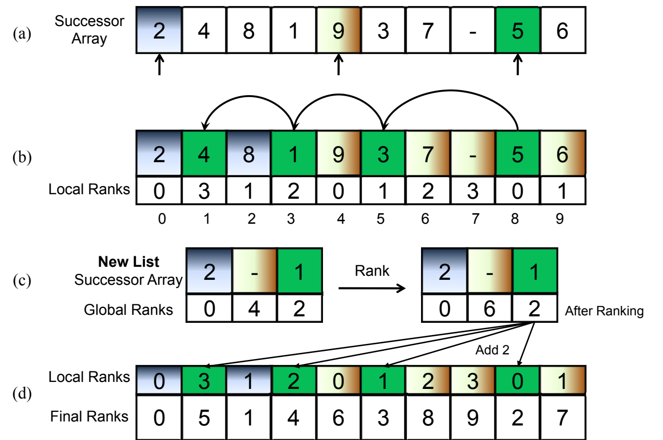


Figure 4: A sample run of the Helman and JáJá list ranking algorithm.

Bader presented implementations of this algorithm on various architectures such as the Cray MTA-2 and Sun enterprise servers [2], and recently with the IBM Cell processor [3], a hybrid multi-core architecture which is often compared to GPUs.

3.2.1 Recursive Helman JáJá Algorithm (RHJ)

The algorithm, as described by Helman and JáJá [12] reduces a list of size n to s and then uses one processing node to calculate the prefix of each node of this new list. It would be unwise to apply the same technique to the GPU, as this will leave most of the hardware underutilized after the first step. Instead, we modify the ranking step of the original algorithm to reduce the list recursively until we reach a list that is small enough to be tackled efficiently by the GPU or by handing over to either the CPU for sequential processing or to Wylie's algorithm on the GPU.

Algorithm 3 lists the recursive version of the Helman JáJá. Each element of array L contains both a successor and rank field. In step 4 of the algorithm, we select p splitter elements from L. These p elements will denote the start of a new sublist and each sublist will be represented by a single element in L_1. The record of splitters is kept in the newly created array L_1. Once all the splitters are marked, each sublist is

Algorithm 3 Recursive Helman-JáJá **RHJ**(L,R,n,$limit$)

Input: Array L, containing the successors of each list element and $n = |L|$
Output: Array R containing the rank of each element in L

1: **if** $n <= limit$ **then**
2: Rank the final list L sequentially
3: **else**
4: Choose p splitters at intervals of $\frac{n}{p}$
5: Define the sublists on L,
6: Copy these splitter elements to L_1
7: Perform local ranking on each sublist of L
8: Save the local ranks of each element of L in R.
9: Set successor pointers for L_1
10: Write sublist lengths in R_1
11: Call Procedure **RHJ**(L_1,R_1,p,$limit$)
12: **end if**
13: Each element in R adds the rank of its sublist from R_1 to its local rank.

traversed sequentially by a processor from its assigned splitter until another sublist is encountered (as marked in the previous step). During traversal, the local ranks of the elements are written in list R. Once sublist has been traversed, the successor information, i.e the element (with respect to indices in L_1) that comes next is recorded in the successor field of L_1. It also writes the sum of local ranks calculated during the sublist traversal to the rank field of the succeeding element in L_1.

Step 6 performs the recursive step. It performs the same operations on the reduced list L_1. Step 1 of the algorithm determines when we have a list that is small enough to be ranked sequentially, which is when the recursive call stops.

Finally, in step 8, we add the prefix values of the elements in R_1 to the corresponding elements in R. Upon completion of an iteration of RHJ, the List R will have the correct rank information with respect to the level in the recursive execution of the program.

The recursive Hellman-JáJá algorithm can be analyzed in the PRAM model [13] as follows. For $p = n/\log n$ sublists, i.e., with p splitters, the expected length of each sublist is $O(\frac{n}{p})$. Hence, the expected runtime can be captured by the recurrence relation $T(n) = T(p) + O(\frac{n}{p})$, which has a solution of $O\left(\frac{\log^2 n}{\log\log n}\right)$. Similarly, the work performed by the algorithm has the recurrence relation $W(n) = W(p) + O(n)$ which has a solution of $O(n)$. So, our algorithm is work-optimal and has an approximate runtime of $O(\log n)$. Notice however that the deviation from the expected runtime for $p = n/\log n$ could be $O(\log^2 n)$, for random lists. The correctness of the algorithm should be apparent from its construction and PRAM style runtime.

4. RHJ IMPLEMENTATION ON THE GPU

Implementing the RHJ algorithm on the GPU is challenging for the following reasons: Programs executed on the GPU are written as CUDA kernels. They have their own address space on the GPU's instruction memory, which is not user-accessible. Hence CUDA does not support function recursion. Also, each step of the algorithm requires complete synchronization among all the threads before it can proceed. These challenges are tackled in our implementation, which is discussed in the next section.

4.1 Implementation in CUDA

All operations that are to be completed independently are arranged in kernels. Global synchronization among threads can be guaranteed only at kernel boundaries in CUDA. This also ensures that all the data in the global memory is updated before the next kernel is launched. Since each of the 4 phases of the algorithm require a global synchronization across all threads (and not just of those within a block) we implement each phase of the algorithm as a separate CUDA kernel. Since we are relying purely on global memory for our computation, CUDA blocks do not have any special significance here except for thread management and scheduling. We follow NVIDIA's guidelines to keep the block size as 512 threads per block to ensure maximum occupancy[18][16].

The first kernel is launched with p threads to select p splitters and write to the new list L_1. The second kernel also launches p threads, each of which traverse its assigned sublist sequentially, whilst updating the local ranks of each element and finally writing the sublist rank in L_1. The recursive step, is implemented as the next iteration of these kernels, with the CPU doing the book-keeping between recursive levels. Finally we launch a thread for each element in L to add the local rank with the appropriate global sublist rank.

A wrapper function that calls these GPU kernels is created on the CPU. It takes care of the recursive step and the recursion stack is maintained on the host memory. CUDA also requires that all the GPU global memory be allocated and all input data required by the kernels be copied to the GPU beforehand. Once we obtain the list size, the required memory image is created for the entire depth of recursion before we enter the function.

The final sequential ranking step (which is determined by the variable $limit$) can be achieved in a number of ways. It can either be handed over to Wyllie's algorithm, or be copied to the CPU or be done by a single CUDA thread (provided that the list is small enough). An appropriate value of $limit$ for each of these scenarios is discussed in the results section.

4.2 Choice of Splitters and Load Balancing

The runtime and work efficiency of this algorithm rests on the proper choice of splitters from the random list. In the crucial local ranking step of the algorithm, each thread will traverse the nodes until it reaches another splitter. Our goal is to make these splitters as equidistant as possible in the actual ranked list, in order to provide all threads equal amount of work. Since we are working with random lists, we cannot make an informed choice of splitters that can guarantee perfect load-balancing among threads in our implementation. This problem is apparent in the multi-threaded Cell implementation[3], but its scale is magnified when this algorithm is implemented in the GPU with tens of thousands of threads and on very large lists.

The original algorithm also calls for $\frac{n}{p\log n}$ splitters to be chosen at random from the list, where p is the number processing elements. With $p \ll n$ for SMP architectures, Helman *et al.*[12] went on to prove that with high probability, the number of elements traversed by a processor is no more than $\alpha(s)\frac{n}{p}$ where $\alpha(s) \geq 2.62$.

The GTX280, however, requires a minimum of 1024 threads per SM to keep all the cores occupied and offset memory fetch latencies from global memory. For our implementation, the assumption that $p \ll n$ no longer holds good as

we have large number of threads that are proportional the list size. For this implementation to succeed we need a good choice of the number of random splitters S such that we are able to get evenly-sized sublists with high probability. For a list of size n and number of sublists p, a few parameters can help us here in deciding the uniformity of the size the sublists: The number of singleton sublists, the standard deviation and the frequency of various sublist sizes.

To understand the impact of these parameters on the runtime, we implemented the RHJ algorithm on the GPU with two choices of splitters: $\frac{n}{\log n}$ and $\frac{n}{2\log^2 n}$. Observe that with $\frac{n}{\log n}$ splitters, the expected number of singleton sublists for a list of 8 M elements is about 16,000. This means that load is not properly balanced as some lists tend to be larger than the others. With $\frac{n}{2\log^2 n}$ splitters, the expected number of singletons is under 10. However, this has the effect of increasing the number of elements in each sublist on the average.

In a particular run on a list of 8 M elements, the largest sublist size with $\frac{n}{\log n}$ splitters is 350 while with $\frac{n}{2\log^2 n}$ the largest sublist has a size of around 10,000 elements. Similarly, the deviation in the case of using $\frac{n}{\log n}$ is higher compared to the deviation with $\frac{n}{2\log^2 n}$ splitters.

Guided by these observations, for large lists we used $\frac{n}{2\log^2 n}$ splitters in the first iteration and used $\frac{n}{\log n}$ splitters in the remaining iterations (of the recursion). One hopes that this would give good performance over $\frac{n}{\log n}$ splitters consistently. However, we observed that there are some inconsistencies. This is likely due to the fact that if one splitter with a large sublist to be ranked locally is in each warp then all the threads in that warp are affected. Moreover, CUDA does not allow user level thread management which means that some of the optimization techniques fail to work in CUDA. Hence, while the performance of RHJ on GPU with $\frac{n}{2\log^2 n}$ splitters is in general good for lists of size 8 M and above, we see small discrepancy for list of 64 M elements.

5. EXPERIMENTAL RESULTS AND DISCUSSION

The various implementations discussed so far were tested on a PC with Intel Core 2 Quad Q6600 at 2.4 GHz, 2 GB RAM and a NVIDIA GTX 280 with 1 GB of on board Graphics RAM. The host was running Fedora Core 9, with NVIDIA Graphics Driver 177.67, and CUDA SDK/Toolkit version 2.0. These algorithms are also compared to the standard $O(n)$ sequential algorithm running on the same PC (**Q6600**)- hereby referred to as the CPU Sequential Algorithm. We test our implementation on both random and ordered lists (as shown in Figure 1).

5.1 Comparison with Similar Architectures

In Figure 5, Wyllie's algorithm shows an interesting pattern. Due to the increased overhead of using 64 bit atomic operations in the same algorithm, we can see that for small lists, the curve is very close to the CPU sequential curve. Unless the list size is at least 512K, Wyllie performs essentially the same as CPU, after 512K, we notice some speedup. However at around 16 M or higher, the performance of the algorithm degrades and falls behind the CPU.

Using the RHJ algorithm on the other hand, shows a clear speedup at about 32 K that increases as the list size in-

List Size	Running Time (msec)			
	CPU Seq.	GPU Wyllie	GPU RHJ $\log N$	GPU RHJ $2\log^2 N$
1 K	0.010	0.050	0.158	0.273
2 K	0.016	0.053	0.185	0.383
4 K	0.033	0.060	0.227	0.493
8 K	0.087	0.084	0.289	0.678
16 K	0.187	0.133	0.340	0.867
32 K	0.607	0.243	0.384	1.11
64 K	1.25	0.441	0.50	1.57
128 K	2.60	1.16	0.86	2.23
256 K	6.57	2.96	1.68	3.50
512 K	19.59	6.68	3.48	4.73
1 M	76.87	19.06	7.27	7.49
2 M	209	41.91	15.29	13.80
4 M	483	89.22	32.23	26.22
8 M	1033	492	150	129
16 M	2139	1694	464	444
32 M	4442	4502	1093	1136
64 M	9572	11013	2555	2561

Table 2: Results of the various implementations on the GPU vs. CPU averaged over multiple runs on random lists.

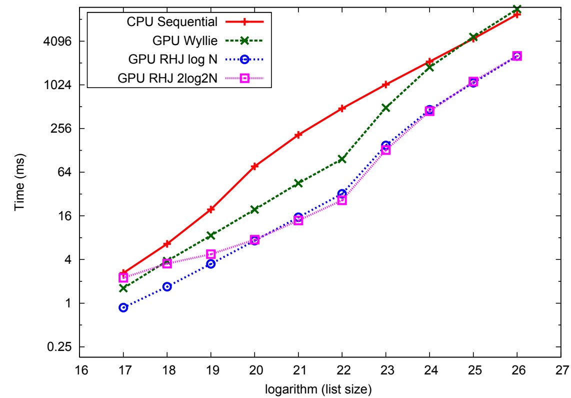


Figure 5: Comparison of the run-times of the list ranking implementations, averaged over multiple runs on random lists.

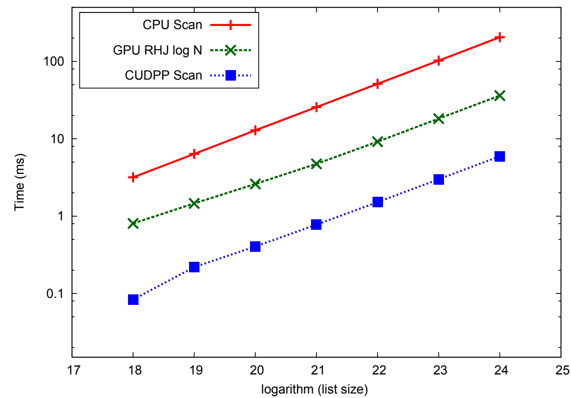


Figure 6: Performance of the RHJ algorithm on an ordered list vs. the implementation for ordered lists from CUDPP [11] and a simple CPU sequential scan.

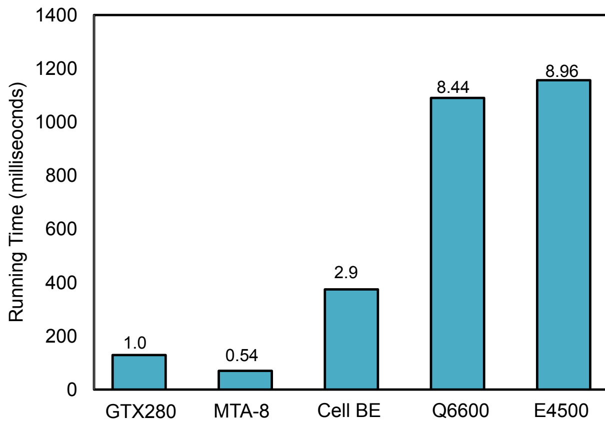


Figure 7: Comparison of list ranking on GTX 280 against other architectures [2, 3] to rank a random list of 8 million nodes. Speedup on a GTX 280 is denoted over the respective bars.

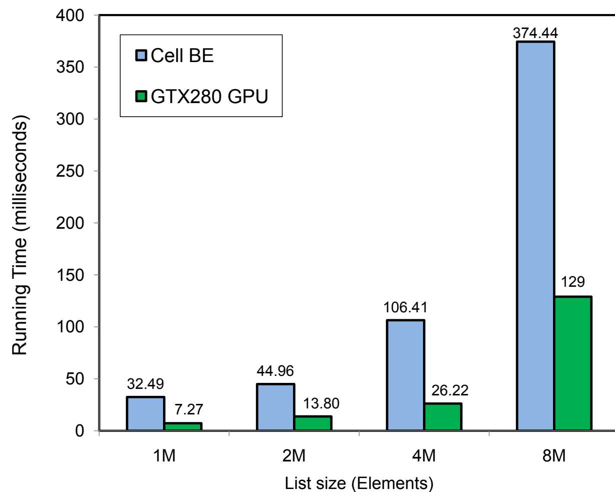


Figure 8: Performance of List Ranking on GTX 280 and Cell BE [3] for random lists.

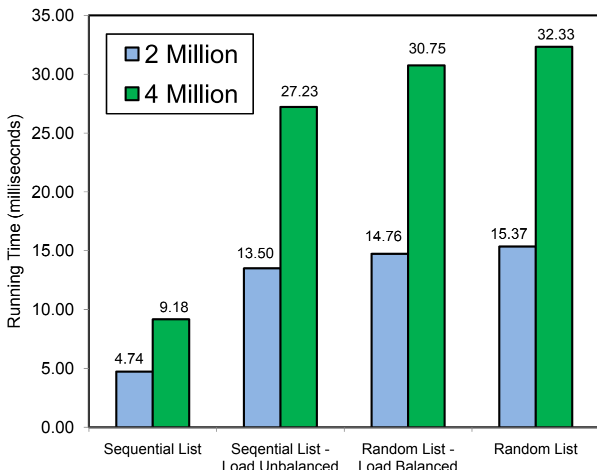


Figure 9: Effects of load balancing and memory coalescing on list ranking for list of size 2 and 4 million nodes

creases. As we recursively go down to a list of less than 32 K, it will be beneficial to handover the final sublist to either Wylie or to CPU for final ranking. It should be noted here that handing over the list to CPU will entail some additional penalties of copying the final sublist data between GPU memory to host memory, but when its sufficiently small, it can be ignored.

The results of implementing RHJ on the GPU clearly shows that dividing the input data into non-overlapping segments and introducing global synchronization (through multiple kernels) will clearly benefit performance as compared to a single kernel that relies heavily on atomic operations to implement critical sections.

In Figure 7, the running time of our algorithm is compared to the following implementations:

1. **Q6600:** Sequential implementation on the Intel Core 2 Quad Q6600, 2.4 GHz with 8 MB Cache
2. **Cell BE:** IBM Blade Center QS20 - on 1 Cell BE processor (3.2 GHz, 512 MB Cache) with 1 GB RAM, Cell SDK 1.1 [3]
3. **MTA-8:** 8 x Cray MTA-2 220 MHz (No data cache) processors working in parallel [2]
4. **E4500** 8 x Sun UltraSPARC 400 MHz processors in a SMP System (Sun Enterprise Server E4500) [2]

GPU vs. Cell BE.

In Figure 8, we provide the performance comparison of the GTX 280 vs. the Cell BE for random lists from 1 million to 8 million nodes. We can see that we have a sustained performance benefit of about 3-4x on these lists. The Cell BE implementation is the most efficient till date and features a latency hiding technique. Also, the Cell has user managed threads and an efficient DMA engine. A fair comparison of the various architectural parameters of Cell vs. GPU is really not possible due to the fundamental difference in their architectures.

5.2 Profile of RHJ on the GPU

List Size	Time for First Iteration (μsec)			Total Time (μsec)
	Split	Local Rank	Recombine	
1 M	11	5350	1094	7182
2 M	15	12273	922	13367
4 M	24	24851	1881	26927
8 M	46	124658	4256	129203

Table 3: Break-up of running time of RHJ on different lists. Total time is the runtime across all iterations.

To understand the reason for the observed runtime, we used the CUDA profiler (provided by CUDA SDK) to breakup the runtime into various phases of the algorithm. Table 3 shows the complete timing breakup required by the 3 phases of the algorithm for the first iteration along with the total runtime. Notice that a large fraction of the total time is spent in the local ranking step of the first iteration. This implies that optimizations to this step can further improve the performance. In a particular run on a list of 8 M elements, as reported in Section 4.2, the largest sublist had

about 10000 elements, while the number of singleton sublists was around 16000. Using the CUDA documentation [16], the approximate runtime can be computed to be about 100 milliseconds, by taking the global memory access times into account. Since the size of the list ranked in subsequent iterations is very small compared to the size of the original list, the combined runtime of all the subsequent iterations is under 1% of the total.

5.3 RHJ on Ordered Lists

We compare our implementation of RHJ on ordered lists with the CUDPP CUDA ordered scan algorithm by Harris *et al.*[19] (Figure 6). Scan works well for ordered lists since it has been heavily optimized and takes advantage of the ordered structure to hide global access latencies through the use of shared memory.

Ordered lists (Figure 1(a)) perform 5-10 times better than random lists on RHJ itself due to perfect load balancing among the threads and the high performance of the GPU of coalesced memory accesses. Note that the algorithm does not know that the lists are ordered. Scan performs about 8-10 times faster than RHJ on orderd lists. Scan operation, however, works only on ordered lists and cannot be used on random lists. A recent paper by Dotsenko *et al.*[8], claims to be even faster than CUDPP scan.

5.4 Irregular Access and Load Balancing

Random lists perform significantly worse than ordered lists using the RHJ algorithm primarily due to two reasons: unbalanced load in the sequential ranking step and irregular memory access patterns due to the random list. On the other hand, Scan [19] has regular and perfectly coalesced memory access as well as even work loads. We studied the impact of access pattern and load balance separately.

We create regular access patterns with unbalanced loads by selecting random splitters on a sequential list. Each thread accesses consecutive elements of the list. The GPU does not use caching on global memory to take advantage of this, however. It coalesces or groups proximage memory accesses from the threads of a half-warp (currently 16 consecutive threads) into a single memory transaction. Coalesced global memory access [16] is a major factor that determines the time taken to service a global memory request for a warp. In the GTX 280, if all the threads of a half-warp access a memory segment of less than 128 bytes, it is serviced with a single memory transaction. In other words, a single memory transaction fetches an entire segment of memory, and a memory transaction devoted to retrieving a single word will lead to wasted memory bandwidth. Regular access pattern can improve the coalescing performance, but not completely.

We create random access with balanced load using a random list, but with splitters pre-selected that are equidistant from each other. This ensures that all threads handle the same number of elements in the sequential ranking step, but the memory access pattern is totally random. The random list is completely uncoalesced as well as load-imbalanced and performs the worst.

Figure 9 shows the effects of load imbalance and irregular global memory access on the performance. Regular memory access even with unbalanced load seems to perform marginally better than irregular access with balanced load. It should be noted that regular memory access pattern does not guarantee full coalescing of memory transactions without perfect splitter selection. Both are, however, much closer to the performance on a random list than the performance on a sequential list. Scan does much better and can be thought of as selecting every other element as a splitter on a sequential list and achieves high performance.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented two implementations of list ranking on the GPU. Wyllies algorithm gives good performance on smaller lists but does not scale well onto larger lists due to its work inefficiency. The Recursive Helman-JáJá algorithm scales well onto larger lists and ranks a random list of 32 million elements in about a second. It outperforms other reported implementations of list ranking on comparable architectures decisively. The RHJ algorithm is best suited for large input lists, so that the massively multi-threaded architecture of the GPU can be fully exploited.

We do run into performance issues with our implementations. The current bottlenecks in the implementations are:

1. In Wyllie's algorithm, the memory operations affect performance significantly, as the algorithm is not work-optimal.

2. For RHJ, memory latency of uncoalesced global accesses and load-balancing are the primary reasons affecting performance.

Our implementation of Wyllie's algorithm implements pointer jumping which may be a primitive useful to various problems. We hope our work will provide some important pointers for implementing other irregular algorithms on massively multi-threaded architectures like the GPU. Some of the lessons learned for implementing irregular algorithms here are:

1. Exploiting massive parallelism of the GPU is key to maximizing performance.

2. Over-reliance on expensive synchronization should be avoided.

3. Some effort in improving load-balancing through probabilistic means of dividing the work among various threads may be helpful in improving performance.

In the future, we hope to implement a hybrid solution to the list ranking problem that distributes the processing power among multiple GPUs and multi-core CPUs.

7. ACKNOWLEDGMENTS

We would like to thank NVIDIA for their generous hardware donations and David Bader (Georgia Tech) and his team for the performance data of the Cell processor.

8. REFERENCES

[1] R. J. Anderson and G. L. Miller. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[2] D. Bader, G. Cong, and J. Feo. On the Architectural Requirements for Efficient Execution of Graph Algorithms. *International Conference on Parallel Processing (ICPP), 2005*, pages 547–556, June 2005.

[3] D. A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE, 2007.

[4] D. A. Bader and G. Cong. A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994 – 1006, 2005.

[5] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein. Evaluating Arithmetic Expressions Using Tree Contraction: A Fast and Scalable Parallel Implementation for Symmetric Multiprocessors (SMPs). In *HiPC '02: Proceedings of the 9th International Conference on High Performance Computing*, Lecture Notes in Computer Science, pages 63–78, London, UK, 2002. Springer-Verlag.

[6] I. Buck. GPU Computing with NVIDIA CUDA. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM.

[7] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix Sums and List Ranking. *Information and Computation*, 81(3):334–352, 1989.

[8] Y. Dotsenko, N. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS)*, pages 205–213. ACM New York, NY, USA, 2008.

[9] N. Govindaraju and D. Manocha. Cache-Efficient Numerical Algorithms using Graphics Hardware. *Parallel Computing*, 33(10-11):663 – 684, 2007.

[10] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM.

[11] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. `http://gpgpu.org/developer/cudpp`.

[12] D. R. Helman and J. JáJá. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. In *ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, Lecture Notes in Computer Science, pages 37–56. Springer-Verlag, 1999.

[13] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[14] G. L. Miller and J. H. Reif. Parallel Tree Contraction and its Application. pages 478–489, Oct. 1985.

[15] A. Munshi. OpenCL Specification V1.0. Technical report, Khronos OpenCL Working Group, 2008.

[16] NVIDIA Corporation. CUDA: Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA, 2007.

[17] M. Reid-Miller. List Ranking and List Scan on the Cray C-90. In *SPAA '94: Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 104–113, New York, NY, USA, 1994. ACM.

[18] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Wen-mei. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204. ACM New York, NY, USA, 2008.

[19] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, Switzerland, 2007. Eurographics Association.

[20] J. Sibeyn. Minimizing Global Communication in Parallel List Ranking. In *Euro-Par Parallel Processing*, Lecture Notes in Computer Science, pages 894–902. Springer, 2003.

[21] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, Ithaca, NY, USA, 1979.