Fast Scan Algorithms on Graphics Processors

Yuri Dotsenko Naga K. Govindaraju Peter-Pike Sloan Charles Boyd John Manferdelli

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
{yurido, nagag, ppsloan, chasb, jmanfer}@microsoft.com

ABSTRACT

Scan and segmented scan are important data-parallel primitives for a wide range of applications. We present fast, work-efficient algorithms for these primitives on graphics processing units (GPUs). We use novel data representations that map well to the GPU architecture. Our algorithms exploit shared memory to improve memory performance. We further improve the performance of our algorithms by eliminating shared-memory bank conflicts and reducing the overheads in prior shared-memory GPU algorithms. Furthermore, our algorithms are designed to work well on general data sets, including segmented arrays with arbitrary segment lengths. We also present optimizations to improve the performance of segmented scans based on the segment lengths. We implemented our algorithms on a PC with an NVIDIA GeForce 8800 GPU and compared our results with prior GPU-based algorithms. Our results indicate up to 10x higher performance over prior algorithms on input sequences with millions of elements.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming.

General Terms

Algorithms, performance.

Keywords

Scan, all-prefix-sum, segmented scan, GPGPU, GPU, parallel algorithm, HPC, many-core.

1. INTRODUCTION

Graphics processing units (GPUs) are programmable processors with high memory bandwidth and high parallelism. They are mainly designed for gaming applications. With the introduction of new features such as atomic and scatter operations, and shared register files, several data parallel algorithms such as *quicksort* [6] can be mapped to GPUs. The basic building blocks in many of these data-parallel algorithms are scan primitives, and several scan algorithms have been designed for parallel processors [4][7].

Recently, many scan algorithms have been implemented for GPUs [11][13][5][6][15]. These algorithms exploit the high memory bandwidth and massive parallelism on GPUs. The current state-of-the-art GPU-based algorithms also exploit shared memory to improve the performance of scans. In this paper, we analyze the issues in mapping scan algorithms to the GPU architecture. We highlight that the prior algorithms deliver suboptimal performance due to high overhead of shared-memory bank conflicts, synchronization, and index arithmetic.

We present fast scan algorithms that map better to GPUs and achieve higher performance than prior GPU-based algorithms. Our main contribution is a novel data representation in shared and global memory that maps better to the GPU memory hierarchy and the scan algorithms. Accesses to the data representation involve no bank conflicts in the shared memory while exploiting the high parallelism on GPUs. Our algorithm involves low overhead compared to prior approaches and the performance of the kernel scales better with shared memory sizes.

We implemented our algorithms on a PC with a modern NVIDIA GPU. We benchmark our algorithms against prior state-of-the-art GPU-based algorithms on several GPUs. Our results on unsegmented scans indicate up to 60% higher performance than prior optimized algorithms. On segmented scans, we observed up to an order of magnitude higher performance over optimized GPU-based segmented scan algorithms.

Organization of the paper: The rest of the paper is organized as follows. In Section 2, we present the related work. In Section 3, we give an overview of scan algorithms and the issues in mapping them to GPUs. We present our scan algorithms and provide analysis in Section 4. In Section 5, we describe our experimental results. We summarize the paper and present future work in Section 6.

2. RELATED WORK

Scan primitive was introduced by Iverson in APL [1]. Blelloch provides extensive overview of scans as building blocks of parallel algorithms and formalizes scan for the PRAM model [4]. Blelloch presented several applications of the scan algorithm such as radix sort [17], sparse matrix vector multiply [16], etc. These algorithms may not map directly to GPUs due to complexities of modern GPU architectures such as memory bank conflicts.

Horn [11] presented the first GPU-based scan algorithm used in the context of non-uniform stream compaction. Horn's implementation utilized graphics programming APIs and used the streaming model for the scan operation. Hensley et al. [12] improved the performance of scan primitives for computing summed area tables. Although these algorithms map well to GPUs, they are not work-efficient. Using graphics programming

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.
Copyright 2008 ACM 978-1-60558-158-3/08/06...$5.00.

APIs, Sengupta et al. [13] and Greβ et al. [18] presented work-efficient scan-based algorithms.

More recently, Harris et al. [5] and Sengupta et al. [6] presented implementations of work-efficient scan and segmented scan primitives, respectively, on NVIDIA GPUs using the CUDA programming API. Their algorithms extend the algorithms proposed by Blelloch [4] to GPUs. Both of these algorithms exploit shared memory on GPUs to achieve higher performance than prior algorithms. However, these algorithms involve bank conflicts and their kernels may not scale well with shared memory size resulting in suboptimal performance.

Chatterjee et al. [7] use a 2D matrix to decompose the input vector into chunks small enough to fit inside a vector register of Cray Y-MP architecture [14]. More generally, Blelloch et al. [19] use loop raking to solve linear recurrences on vector architectures. Our algorithm is similar to these approaches. As we use a shared register file on the GPUs, our mapping accounts for memory bank conflicts and overheads of index arithmetic and synchronization. Moreover, we generalize their representation to a multi-dimensional matrix representation that maps better to the memory hierarchy on GPUs.

3. OVERVIEW OF GPUS AND SCAN PRIMITIVES

In this section, we provide a brief overview of scans and the GPU programming model. We highlight some of the main issues in mapping the prior algorithms to GPUs.

3.1 Scans

The *scan* primitive [1] operates on a monoid M with an associative binary operation $\oplus$: $M \times M \rightarrow M$ and a left identity element $\varepsilon_{\oplus} \in M$, such that $\forall a \in M$, $\varepsilon_{\oplus} \oplus a = a$. Given an input sequence $A=[a_0, \ldots, a_{n-1}]$ of n elements, the exclusive scan primitive transforms A into output sequence $B=[b_0, ..., b_{n-1}]$ such that $[b_0, ..., b_{n-1}]=[\varepsilon_{\oplus}, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus \ldots \oplus a_{n-2}]$. The inclusive scan primitive transforms A into $[a_0, a_0 \oplus a_1, \ldots, a_0 \oplus \ldots \oplus a_{n-2} \oplus a_{n-1}]$. In this work, we focus on exclusive scans, noting that the ideas can be trivially extended to handle inclusive scans. The described primitives are forward scans. The backward scan primitives are similar to the equivalent forward scans, but traverse the input sequence in reverse direction. The exclusive backward scan transforms A into $C=[\varepsilon_{\oplus}, a_{n-1}, a_{n-1} \oplus a_{n-2}, \ldots, a_{n-1} \oplus \ldots \oplus a_1]$.

Examples of binary operations used in scans include addition, multiplication, minimum, and maximum operations. These binary operations work on floating point or integer operands. The identities used in scans are 0, 1, $+\infty$, $-\infty$, respectively. In this paper, the exact operation and data types have no difference on our algorithm, so we use + as the operation of choice in the rest of the paper and refer to this scan primitive as +-scan. An example of the +-scan application on an input array A is given below. The output arrays for forward and backward scans are shown in B and C, respectively

```
A = [1, 7,-4, 2, 2,-1, 5]        // input
B = [0, 1, 8, 4, 6, 8, 7]        // exclusive, forward
C = [0, 5, 4, 6, 8, 4, 11]       // exclusive, backward
```

The scan primitive operates on the entire sequence. In practice, many applications need to scan several sequences. Rather than executing several independent scans, one for each input sequence, it has been a common practice to use the *segmented scan* primitive [2]. The input sequences, called segments, are stored together, one after another, in one input vector. To scan these sequences simultaneously, the segmented scan primitive needs an additional input – a vector that enables scan to identify original subsequences. For instance, this information can be conveyed by a vector of head-flags where a set flag denotes the first element of a new segment. Similar to scans, the segmented scan primitives can be exclusive and inclusive as well as forward and backward. We focus on exclusive forward and backward segmented scans. If segments are identified via head-flags, the backward segmented scan must treat set flags as the segment end flags. An example of the segmented +-scan application is shown below:

```
A     =[1, 7,-4, 2, 2,-1, 5]     // input
Flags =[1, 0, 1, 1, 0, 0, 0]     // head-flags
B     =[0, 1, 0, 0, 2, 4, 3]     // exclusive, forward
C     =[0, 5, 4, 6, 0, 0, 7]     // exclusive, backward
```

3.2 GPU Programming Model and Scans

GPUs consist of a number of multiprocessors, each of which can execute the same program on each element of a data set. For optimal performance, the data set is typically decomposed into fixed-size blocks that can each be assigned to a multiprocessor core. Each multiprocessor processes the fixed-size blocks by executing on a small subset of the data elements in a block simultaneously. Multiprocessor context switches between subsets of data elements are inexpensive compared to CPU threads. By switching the multiprocessor to a different subset of data elements, pipeline stalls due to data hazards and memory latency to the DRAM can be effectively hidden. We refer to a group of threads executed in SIMD fashion as a *warp*.

Recent GPUs such as the NVIDIA G80 have a shared memory or register file with synchronization primitives to enable communication between threads running on the same thread block. The shared register file typically consists of several memory *banks* that can be accessed simultaneously by threads of a warp. The accesses to shared register file are similar to local register accesses except when the memory accesses involve register bank conflicts. A conflict occurs when several threads access the same shared memory bank; in which case, the accesses are serialized and their latency increases, resulting in degraded performance.

The performance of GPU memory subsystem depends on memory access patterns. The best performance is achieved when memory accesses are *coalesced*; *i.e.*, *k* consecutive threads of a warp reference *k* properly aligned contiguous elements of the same size.

Scan primitives can be efficiently implemented by exploiting the large number of threads on GPUs and using shared memory to reduce the latency of memory accesses. Many of the current GPU-based algorithms [5][6] use a traditional binary-tree-based algorithm to improve the performance of scans.

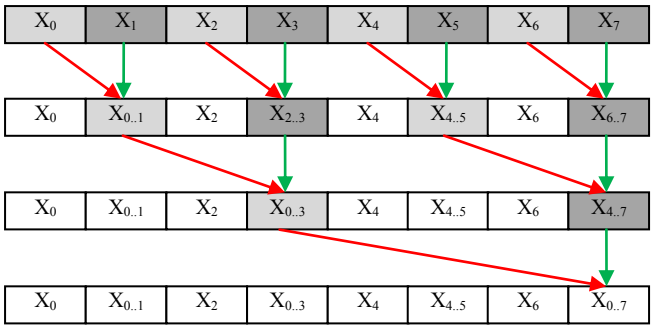

Figure 1. Tree-based reduction of an 8-element sequence.

The binary-tree-based scan algorithm proceeds in *log n* stages. In each stage, the binary operator is applied to two distinct elements in the input array and the output is stored into a new array as shown in Figure 1, where different shading denotes elements accesses by threads of the same warp. In order to further improve the performance, Harris et al. [5] proposed the fast-scan algorithm. The fast-scan algorithm decomposes the input array into blocks of size *B* that fit in the shared memory and performs scan on the decomposed blocks. The scan on the entire sequence is then implemented using a recursive multi-block scan.

In order to utilize the high parallelism on GPUs, the fast-scan algorithm performs an efficient mapping of the threads on the multi-processors to the elements in the stages of the binary tree. In each stage, a thread operates on two distinct elements and the output is written to a shared-memory location. The fast-scan algorithm also uses a padding scheme, based on the number of shared memory banks, to improve performance of logical accesses to shared-memory locations. The additional padding reduces bank conflicts in accessing the elements in shared memory. The resulting algorithm is fast and achieves significantly higher performance than prior GPU-based unsegmented scan algorithms. Sengupta et al. [6] proposed segmented scan algorithms which use a tree-based technique, similar to those of the fast-scan, for arrays with multiple segments. Although the algorithms are fast, their mappings on GPUs have several issues:

- **Global memory accesses:** The fast-scan algorithm writes the results of intermediate scans in shared memory to the global memory. While this reduces the computation, it has a higher global memory overhead than necessary in a streaming architecture.
- **Shared memory accesses**: The padding scheme in fast-scan eliminates the bank-conflicts in the first few stages. However, the fast-scan algorithm has high-degree bank conflicts in the higher stages whenever $B>numbanks^2$, where *B* is size of the block being scanned. This is mainly because the stride for memory accesses in the higher-stages becomes a multiple of the number of shared memory banks. This may limit the performance and scalability if shared memory size increases faster than the number of banks in future architectures. In addition, the padding scheme recomputes element indices for each level of the tree per thread warp active on the level, introducing additional overheads; thus, the overheads increase with the number of threads.
- **Synchronization between stages**: In order to communicate between threads across stages, intermediate values after a stage is complete are written to shared memory and threads are synchronized before beginning the next stage. Using a binary-tree based algorithm, O(*log n*) synchronizations are required. The synchronization operations in the algorithm can be expensive as the number of threads increase.

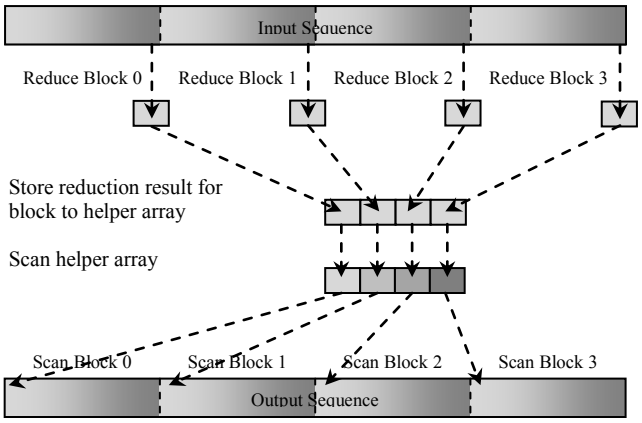

Figure 2. Two-level recursive scan for four blocks.

In this paper, we address these problems using a memory-efficient algorithm that achieves better performance than prior GPU-based scan algorithms.

4. SCAN ALGORITHMS

In this section, we present our algorithms for unsegmented and segmented scans. We later extend our algorithms to handle backward scans. We present an analytical analysis of our algorithms.

4.1 Unsegmented Scan

Given an array A_0 of size N, we decompose the array into blocks of size B that fit in shared memory. We reduce the elements in each block and store the result of the reduction into a new, smaller array A_1 of size $\lceil N/B \rceil$. We recursively invoke the scan operation on A_1. We stop the recursion when the size of the input array to scan is no greater than B. We then perform a scan operation on A_1 and store the result in A_1. Finally, we perform a scan operation on each block B_i of A_0 and combine the element $A_1[i]$ with elements in block B_i. We store the result in A_0, which is the scan of array A_0. This is schematically shown for two levels in Figure 2. Each level of recursion involves synchronization[1] to ensure that the next stage uses proper values produced on the prior stage. We can abstract the array A_0 as a BxBx…xB matrix, and the recursive invocation as the application of our algorithm to each dimension in the matrix.

We refer to our algorithm as *MatrixScan*. The main component of *MatrixScan* that is crucial for achieving high performance is the GPU kernel shown in Figure 4 to scan a block efficiently. The reduction kernel and the recursion logic are rather obvious and omitted from the discussion. Our algorithm generalizes the binary-tree based scan algorithm proposed by Blelloch [4] for vector architectures and maps it efficiently to the memory hierarchy on recent GPUs. Both the reduction and scan operations

[1] On today's GPUs, the commands to execute each level of the multi-block scan recursion are issued from the host; thus, the synchronization between the levels is implicit.

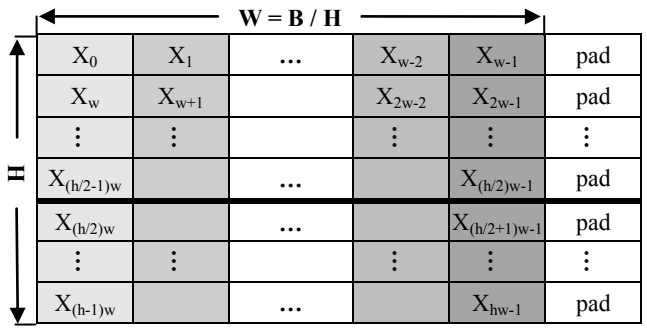

Figure 3. Sequence arranged as a 2D matrix.

on the blocks of an array can be performed in shared memory. Our shared memory algorithm is very similar to our multi-block scan algorithm. We arrange the block in the shared memory as a 2D matrix with dimensions $W \times H$, as shown in Figure 3.

Our shared memory algorithm proceeds in three steps, as shown in Figure 4. First, H threads work in parallel; each thread sequentially reduces the row corresponding to the thread ID in the matrix and stores the result into the auxiliary array `column`. We then synchronize the threads to ensure that the values are written to the auxiliary array. The second step scans `column` using several threads for best performance; this produces scan results p_r of subsequences preceding each row r. Third, H threads work in parallel; each sequentially scans the corresponding row, initializing the running scan result `res` to p_r, thus propagating p_r across r.

We pad W such that the row length of the matrix is prime relative to *num_banks* in the shared memory. Our choice of row length ensures that there are no bank-conflicts while accessing shared memory. Furthermore, in order to utilize the parallelism on the GPU, H is a multiple of the GPU's warp size. Since a sequential scan is performed on each row and involves no synchronizations, the performance of *MatrixScan* operation can be further improved by maximizing W. We therefore, assign H to be the warp size in our implementation. Our choice of H and W ensures high granularity of work per active thread between the synchronization points (steps one and three).

In step two, the *column* can be scanned by multiple threads of a warp similarly to *MatrixScan*. We logically represent the *column* as a 2D matrix $Wc \times Hc$, mask Hc threads to work on Wc-long rows (steps one and three), and sequentially scan the Hc-long column in step two. We note that while this gives the best performance, the gain may not be significant as compared to scanning the entire column sequentially using a single thread.

If the block size is less than the matrix size, then we pad the remaining elements to the identity element. This approach may result in extra computation for the last, partial block, but simplifies the logic of *MatrixScan* for scanning a full block. The resulting kernel has lower instruction count and uses fewer registers.

4.1.1 Backward Scan

The backward scan primitive is analogous to forward scan, but traverses the input sequence in the reverse direction. There are two approaches to implement a backward scan. First, when a block is loaded into shared memory, it is reversed, so that the

```
MatrixScan() {
  // Step I. Reduce rows using H threads.
  if (threadId < H) {
    T* row = &s[threadId*(W+pad)];
    T res = row[0];
    for (int i=1; i<W; ++i) res=res⊕row[i];
    column[threadId] = res;
  }
  sync();
  scanColumn();     // Step II.
  sync();
  // Step III. Scan rows using H threads.
  if (threadId < H) {
    T* row = &s[threadId*(W+pad)];
    T res = column[threadId];
    for (int i=0; i<W; ++i) {
      T t = row[i];
      row[i] = res;
      res = res ⊕ t;
    }
  }
}
```

Figure 4. Matrix-based unsegmented forward scan kernel.

forward scan algorithm can be applied; when the result is stored into memory, it is reversed again. This approach may introduce some overhead for sequence reversal, but allows one to reuse more code of the forward scan implementation. Alternatively, the computation of the forward scan can be reversed without the need to reverse the sequence, avoiding potential overhead. Our implementation uses the latter approach. Our backward scan is similar to forward scan except in the traversal; we reverse the loops and the order of $\oplus$ operands to support non-commutative $\oplus$-operations in *MatrixScan*. Our backward scan implementation achieves performance comparable to our forward scan for all inputs.

4.1.2 Analysis of Matrix-based Scan Kernel

Our algorithm requires lower memory bandwidth than prior scan algorithms on GPUs. For example, prior GPU-based algorithms require $\approx 4N$ memory accesses and our algorithm reduces this number to approximately $3N$ memory accesses. Our matrix representation eliminates bank-conflicts in accessing shared memory for both the reduction and scan phases. Moreover, the sequential reduction and scan algorithms on the rows in shared memory require only a single conditional statement and avoid the additional index arithmetic of the tree-based algorithms. In addition, the sequential operations on the rows do not require synchronization operations. The resulting algorithm has significantly lower overhead than binary-tree-based algorithms.

Our kernel algorithm also scales better whenever $B > num\_banks \times warp\_size$. Therefore, as the shared memory size increases, the algorithm performs better on larger blocks. In contrast, the binary-tree based algorithms perform well when $B < numbanks^2$ and exhibits high-degree bank conflicts for larger B.

The recursive scheme of our or prior scan algorithms requires one reduction value per block to be stored to and later loaded from GPU memory to be propagated across the corresponding block. As we access a single element per block, the accesses are uncoalesced and the performance is usually lower than that of coalesced accesses. However, the number of such uncoalesced

accesses is much smaller as compared to the total number of memory accesses in the entire scan algorithm. The recursive level k performs only $2(N/B^k)$ uncoalesced accesses. For example, for a sequence of 1M elements and a block size of 1K, the number of uncoalesced accesses is 2*1M/1K=2K, while the total number of accesses is 3*1M+2K; thus, the ratio is approximately 6.5e-4. In practice, our experiments showed that these memory accesses contribute to no more than 2% to the total runtime over the input sizes.

4.2 Segmented Scan

We extend our matrix-based formulation of unsegmented scans on GPUs to segmented scans. We use a 2D matrix with padding to arrange a block, as described in Section 4.1. A partial block is padded in shared memory with $\varepsilon_{\oplus}$ and scanned as a full block. We use a compressed head-flag representation to indicate the start of a new segment. Each element has a bit-flag associated with it. To reduce the shared memory usage, 32 flags of consecutive elements are packed into an integer. When sequentially scanning a piece of a sequence, the running scan result must *not* propagate to the next segment; instead, it must be reset. This is easy to incorporate into the sequential code traversing rows (or `column`). Figure 5 shows the segmented scan kernel pseudocode for a block of an input sequence (first recursive level)[2].

The algorithm consists of three stages. During the first stage, each of H threads scans its row of the matrix. If the start of a new segment is detected by the statement labeled 1, the running scan result `res` is reset to $\varepsilon_{\oplus}$, starting a new scan. The scan result for the row's last element is saved in `column`; the row flag `f` is saved in `flagColumn`. Arrays `column` and `flagColumn` are used in the second step of applying segmented scan to compute preceding scan values for the *first* segment of *each* row. Note that head-flags in `flagColumn` are not compressed. Each column flag occupies a 4-byte word w. This is an affordable space overhead that allows us to maintain a highly-parallel implementation. We use several threads to perform the segmented scan of `column` for the best performance, similarly to how it is done for the unsegmented scan. In addition, the code must be extended to not propagate the running scan value beyond the start of a new segment. Finally, the scan value p_r of the subsequence preceding each row must be propagated across the first segment of the row if the segment started before the row's first element. We use variable `ff`, initialized to 0, to detect the start of a new segment. Statement 5 true-branch propagates p_r only until a set flag is encountered.

4.2.1 Backward Segmented Scan

The backward segmented scan is similar to the forward segmented scan primitive, but traverses the sequence from end to start. Clearly, it is prudent to have a unified flag representation for both forward and backward segmented scans. We use the compressed head-flag representation, so that, for a forward segmented scan, a set flag denotes the start of a segment. However, for the backward scan, a set flag denotes the *end* of a segment. There are two choices to implement a backward segmented scan kernel. One can reverse the data and flags and use the forward segmented scan algorithm as suggested in [6]. We reverse the computation and perform the backward segmented scan similar to the unsegmented backward scan algorithm in Section 4.1.1.

[2] The code for recursive levels in multi-block segmented scan is slightly different and is discussed in Section 4.2.2.

```
MatrixSegScan() {                    // for W ≤ 32
  // Step I. Scan rows using H threads.
  if (threadId < H) {
    T* row = &s[threadId*(W+pad)];      // thread row
    FlagT f = load thread flag;         // thread flag
    T t = ε⊕, res = ε⊕;
    for (int i=0; i<W; ++i) {
      // if i-th flag set, reset res
1:    res = (f&(1≪i))==0? (res ⊕ t): ε⊕;
2:    t = row[i];
3:    row[i] = res;
    }
    column[threadId] = res;
    flagColumn[threadId] = f;
  }
  sync();
  scanColumn();                      // Step II.
  sync();
  // Step III. Fix rows using H threads.
  if (threadId < H) {
    T* row = &s[threadId*(W+pad)]; // thread row
    FlagT ff = 0, f = load thread flag;
    T v = column[threadId]; // value preceding row
    for (int i=0; i<W; ++i) {
4:    ff |= f & (1≪i); // 0, if fixing row's 1st segment
5:    if (ff == 0) { // propagate prec. value in 1st segment
        row[i] = v ⊕ row[i];
      }
    }
  }
}
```

Figure 5. Segmented scan kernel.

4.2.2 Multi-block Recursive Segmented Scan

To support efficient segmented scans of large sequences, it is necessary to divide the input into blocks that can be scanned inside shared memory. We adapt a recursive solution similar to that of unsegmented scan (see Figure 2); we implemented the scan-recursion-propagate (sRp) approach with two additional optimizations discussed below. For each block b, we collect r_b, the result of scanning b's last segment, as well as the block flag f_b, computed by OR-ing all flags corresponding to b's elements. Recursive level flags are not compressed; each flag occupies a 4-byte word. We trade acceptable space overhead to maintain efficient parallelization by enabling multi-processors to write f_b to non-overlapping memory locations.

A set f_b flag indicates that a new segment starts inside b and the scan result of the subsequence preceding b must not be propagated to blocks following b in the sequence. r_b and f_b are inputs to the next recursive level of multi-block segmented scan. Pseudocode in Figure 5 must be slightly changed to produce proper segmented scan result for recursive levels. The statement labeled 1 must reset `res` to `t`, rather than $\varepsilon_{\oplus}$, because a set flag on recursive levels does not indicate the start of a new segment; it merely indicates that the segment ends somewhere inside the block. For the same reason, statement 1 must be moved right after statement 3 and statements labeled 4 and 5 must be swapped.

For each block b, the recursive call produces p_b – the scan result of the subsequence of the segment preceding the first element of

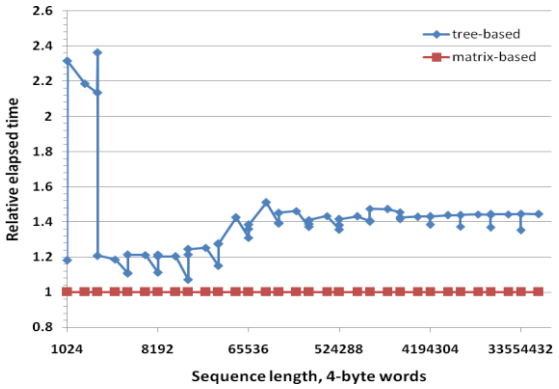

Figure 6. Relative runtime of forward scans.

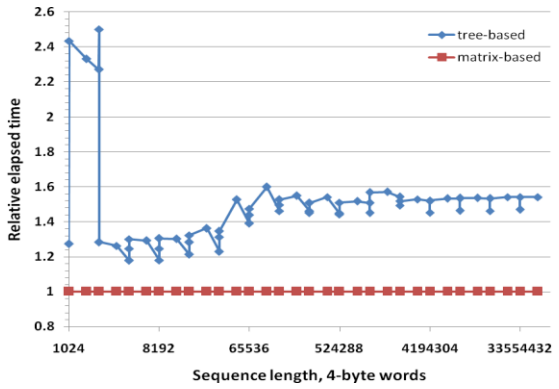

Figure 7. Relative runtime of backward scans.

b. The value of p_b must be propagated across the first segment s of block b, if s starts before b and continues in b. During the first scan phase, we efficiently compute the length of the first block segment s_b, using a scan-like algorithm, and save[3] it along with r_b and f_b. The following propagation phase reads s_b and r_b and fixes only the first s_b elements of b by $\oplus$-ing them with r_b. Knowing s_b enables us to perform another optimization for blocks that lie entirely within one segment. For such blocks, we perform a faster unsegmented scan producing equivalent output.

5. EXPERIMENTAL EVALUATION

We have implemented our matrix-based algorithms for the forward and backward segmented and unsegmented scans using the CUDA programming API [9]. We studied their performance on a PC with a high-end Intel CPU and an NVIDIA GPU running Windows XP. Unless otherwise specified, we report results for an NVIDIA 8800 GTX GPU, since the conclusions are the same for all GPUs that we tried.

We have measured the performance of our implementation and compared it to current state-of-the-art implementations on GPUs. We measured times to complete a computation on the GPU assuming that input and output sequences reside in GPU memory. Our timings did not include the data transfer times between the host and GPU memories since the scan primitives are usually used as intermediate steps of a larger computation performed on data located in GPU memory. To obtain more accurate timing, we average runtimes of several consecutive invocations with the same parameters.

5.1 Unsegmented Scan

To achieve the best performance, we code our matrix-scan kernel carefully. We unroll loops wherever possible to reduce the overheads of indexing and evaluating conditionals. We select B to be a power-of-two for two reasons. First, this gives us coalesced memory accesses when reading/writing the sequence. Second, this allows us to use shifts for index arithmetic instead of more expensive multiplies. While only one warp of threads is used to perform computation, we use several warps to access GPU memory efficiently. On modern NVIDIA GPUs such as GeForce 8800 GTX, the block size of 1K four-byte elements and the thread block of 256 threads give us the best performance for our implementation.

We compared the performance of our novel, matrix-based scan with an optimized GPU implementation available with the CUDPP library [9]. The CUDPP scan primitive uses a binary-tree-based kernel as described in Section 3 and incorporates several optimizations including register caching. Another tree-based implementation of scan, distributed with the NVIDIA CUDA SDK [8], showed at least two times lower performance compared to CUDPP's scan due to higher overhead of synchronization, address computation, and bank conflicts; thus, we do not report results for CUDA SDK scan.

We measure runtimes for sequences with input lengths 2^k-1, 2^k, 2^k+1, 2^k/2*3+1 4-byte word input lengths, where k=10,...,25. We performed our experiments on both non-powers-of-two and power-of-two array sizes, since implementations on general inputs may not be as optimized as powers-of-two arrays. In our experiments, we used +-scans on sequences of `float` elements. The performance of other supported operations such as *, min, max and other input data types such as `int` is similar with the exception of integer multiplication since this is a more expensive operation on current GPUs.

Figures 6 and 7 show relative elapsed time of CUDPP scan and matrix-based scan executions for forward and backward scans, respectively. The runtime is computed relative to that of the matrix-based version, whose performance is 1.0. For sequences larger than 1M elements, the matrix-based scan shows up to 52% (59%) higher performance compared to that of the forward (backward) tree-based scan. The performance difference between the algorithms for power-of-two size sequences tends to be smaller because CUDPP's scan uses a specialized algorithm for loading/storing full blocks. The runtimes, measured in milliseconds for a 1M sequence, are 0.418 (0.438) and 0.298 (0.302) for the forward (backward) CUDPP and matrix-based scans, respectively. For 32M sequences, the runtimes are 11.50 (12.55) and 8.5 (8.55) ms, respectively. Our implementation is able to handle large input arrays by using additional computation to virtualize the blocks on the GPU, while CUDPP is limited to at most 64M-1K elements due to current hardware limitations on GeForce-series GPUs.

Runtimes for short sequences are fractions of a millisecond and are primarily dominated by the cost of launching a kernel on the

[3] We store s_b and s_f together in a 4-byte word to save space. w's highest bit is set to 1 if f_b is not zero.

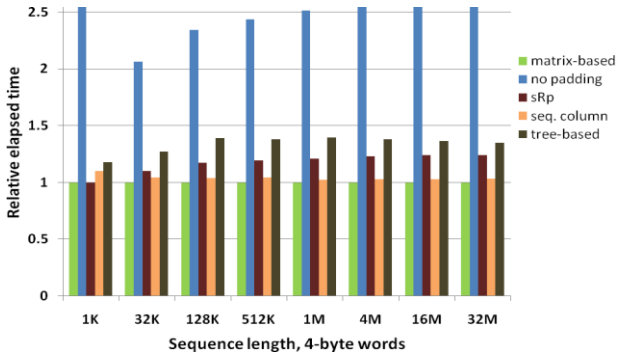

Figure 8. Effect of optimizations in matrix-scan.

GPU and loading/storing data, and less so by the computation time, so it is hard to make accurate comparison of kernels. For a 1K-block scan, the matrix-based kernel shows 18% (27%) improvement over the tree-based kernel for the forward (backward) direction. There is an interesting spike in relative performance for lengths between 1K+1 and 2K. The matrix-based scan runs more than two times faster. The reason is that the matrix-based kernel handles up to 2K-blocks, while CUDPP's kernel works only for 1K-blocks. Therefore, the matrix-based scan performs a single scan of a 2K-block, and the tree-based scan has to recursively scan two 1K-blocks, making two passes over the sequence. Note that this limitation might impede the performance of CUDPP's multi-scan for a typical case of scanning a 1920x1080 image.

Figure 8 shows the effect of disabling one of the optimizations in our implementation of the forward scan; the runtimes are normalized to that of the matrix-based scan with all optimizations, denoted as *matrix-based*. The performance of CUDPP's scan is denoted as *tree-based* and given only for completeness. The *no padding* series denotes the performance of matrix-based scan without row padding and shows the effect of having shared memory bank conflicts. The performance of *no padding* is up to 2.6x worse than that of *matrix-based*, which indicates that bank conflicts may significantly degrade performance. The *sRp* (scan-recursion-propagate) series corresponds to the performance of scan without our reduction optimization, which, for large sequences, is up to 24% slower than matrix-based scan. This matches with our theoretical analysis of matrix-based scan since it has 25% less memory accesses. Note that our algorithm without reduction optimization also outperforms CUDPP due to our efficient matrix-based scan kernel. Finally, *seq. column* corresponds to performance of matrix-based scan where the column is scanned by one thread; this may result in up to 4.5% performance degradation.

5.2 Segmented Scan

To the best of our knowledge, the only segmented scan implementation on GPUs is described in [6]; it is a tree-based algorithm. Execution times are available for several power-of-two lengths on an NVIDIA GeForce 8800 GTX. We compare the performance of this implementation with that of our matrix-based segmented scan.

Our segmented scan implementation is slightly slower than our unsegmented scan implementation due to additional logic. For a given input length, the performance of our algorithm depends on the lengths and layout of segments, due to the two optimizations presented in Section 4.2.2. The matrix-based segmented scan is only up to two times slower than the corresponding unsegmented scan for the worst segment configuration and only up to 1.55 times slower for the best segment configuration. Because its performance differs very little for two input lengths l_1 and l_2 such that $l_2=l_1+1$, we present results only for sequences of 2^k lengths.

Figures 9 and 10 show relative execution times for forward and backward segmented scans. Each chart has three versions: *matrix-best* and *matrix-worst* denote best and worst performance of the matrix-based segmented scan, respectively; *tree* denotes the performance of the tree-based implementation from [6]. Runtimes are relative to those of the *matrix-best* variant.

During the scan phase, we compute and store the length of the first segment for each block. The best performance is achieved if each block b starts with a new segment, because the second phase does not have to propagate the scan result p_b of the sequence preceding b across b. The worst performance is achieved if the sequence is composed of a block-long segments ending at the last element of each block, because the propagation phase has to propagate p_b across the entire b. The runtime for other segment configurations is between the best and the worst cases. The *matrix-best* variant outperforms *matrix-worst* by up to 27% (25%) for the forward (backward) segmented scan.

When a sequence is composed of a few segments that span several blocks, we perform faster unsegmented scan for blocks falling

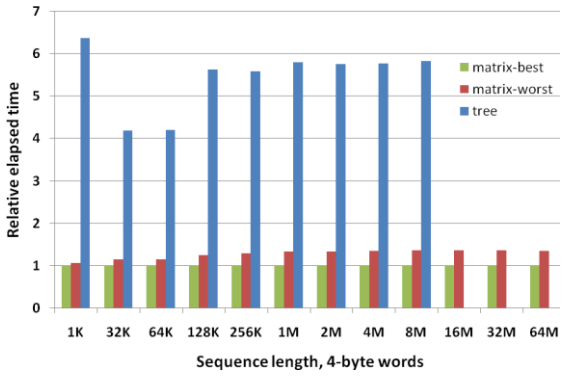

Figure 9. Relative runtime of forward segmented scan.

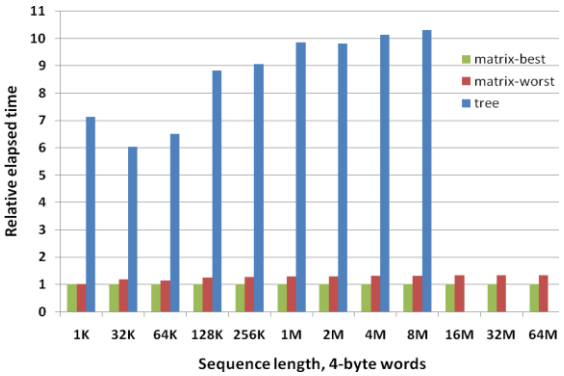

Figure 10. Relative runtime of backward segmented scan.

within one segment. The best case performance for such a sequence is achieved when there is only one segment. For large one-segment sequences, we observed that enabling this optimization improves performance by up to 24% (18%) for forward (backward) segmented scans. Compared to *matrix-best*, the performance is up to 4% (8%) worse, because the propagation phase does the most work propagating *p* across entire *b*.

Compared to the tree-based implementation *tree*, our segmented scan demonstrated much better performance. For input sequences of 1M+ elements, *matrix-worst* shows up to 4.35 times better performance than *tree* does for the forward segmented scan, and up to 7.85 times better performance for the backward segmented scan. *Matrix-best*'s performance is up to 5.84 times better than that of *tree* for the forward segmented scan, and up to 10.32 times better for the backward direction.

5.3 Radix Sort Application

Many applications [4][7][10][12] that use scan primitives as building blocks for parallelization will benefit from our fast matrix-based scan implementations. The overall performance improvement will be proportional to the fraction of total time in executing scans. To test the impact of faster scans on real applications, we integrated our matrix-based scan implementation with CUDPP's global radix sort [9], based on the algorithm in [10] and described in detail in [5].

Radix sort sorts integer keys by processing individual bits. CUDPP implements a parallel radix sort via invocations of three GPU kernels for each bit of the key, beginning from the lowest to highest. An integer array *a* of length *n* is sorted in the ascending order. The first phase constructs a temporary array *t* of length *n*, in which $t[i]=0$ iff $a[i]$'s bit is 1, and $t[i]=1$ iff $a[i]$'s bit is 0. *t* is the input to the forward unsegmented +-scan that, for each element *i*, computes the number of 0-bit elements to the left of *i*. Finally, the third stage uses the output of scan to scatter the elements to proper locations.

The execution time of the first two stages is independent of input; however, the performance of scatter depends on memory access patterns. Scatter takes the least time if all elements are the same, because scatter writes are coalesced. Similarly, scatter takes the most time when none of the writes are coalesced, *e.g.*, for input $a[i]=i$, $i=0..n-1$. The fraction of scan execution time is 30-35% of the total sort time when the scatter takes the most time and ~55% when the scatter takes the least time.

We replaced CUDPP's scan by our faster scan in CUDPP's radix sort. For larger sequences[4], our scan boosts the overall performance of the radix sort by 9.5–11.6% for the worst-performance scatter, and by 12.5–17.7% for the best-performance scatter; this is consistent with the expected speedups.

5.4 Performance of Scan across GPUs

Scans have low computational intensity – therefore, their performance is highly influenced by available memory bandwidth. We measured performance of the scan primitives on three NVIDIA GPUs: 8800 GTX, 8800 GT, and 8600 GTS, which have theoretical peak bandwidth of 86.4 GB/sec, 57.6 GB/sec, and 32 GB/sec, respectively. Figure 11 shows the effective memory bandwidth achieved by our implementation of scan primitives on

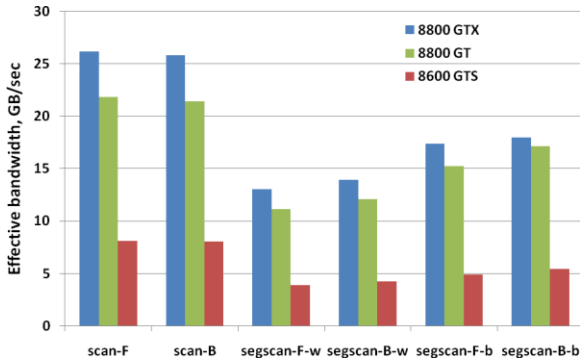

Figure 11. Effective scan bandwidth for 1M sequence.

a 1M-element sequence of floating point data; *F* and *B* denote the forward and backward directions, *w* and *b* denote worst- and best-case segmented scans. We compute the effective bandwidth as *n**2*sizeof(float)/*t*, where *n* is sequence length, 2 is the number of read/write operations per element in a sequential scan, and *t* is run-time. As expected, the performance of the scan primitives increases with the increase in memory bandwidth, achieving the best performance on an 8800 GTX. In comparison to the current DRAM bandwidth available to CPUs, the GPUs have an order of magnitude higher memory bandwidth. Therefore, the performance of GPU-based scans of large sequences outperforms their CPU-based counterparts significantly.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a novel matrix-based scan algorithm. We have presented a detailed analysis on the mapping of our algorithm to GPUs and contrasted it with prior GPU-based scan algorithms. We have highlighted its advantages and extended it to segmented scan and backward scan algorithms. We have also compared our results to the state-of-the-art algorithms. In practice, our results have demonstrated a significant performance improvement over prior GPU-based algorithms.

As a part of the future work, we would like to perform experiments and study scalability issues on newer generation GPUs and other highly-parallel architectures. We would also like to explore new hardware extensions that improve the programmability on GPUs and apply them to scan primitives.

7. ACKNOWLEDGEMENTS

We would like to thank Burton Smith, Brandon Lloyd, and other members of the many-core incubation for useful discussions and feedback. We thank Craig Mundie for the support. Many thanks to Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John Owens for providing run-times of their segmented scan implementations. We would like to thank Henry Moreton for hardware support.

8. REFERENCES

[1] Iverson, K. E. A Programming Language. Wiley, New York, 1962.

[2] Schwartz, J. T. Ultracomputers. ACM Transactions on Programming Languages and Systems, 2(4), pp 484-521, Oct. 1980.

[4] The improvement is even higher for small sequences.

[3] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, v. 1.1, Nov. 2007, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[4] Blelloch, G. E. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990. Book Chapter in Synthesis of Parallel Algorithms, Reif, J. H. (Ed.).

[5] Harris, M., Sengupta, S., and Owens, J. D. Parallel Prefix Sum (Scan) with CUDA. GPU Gems 3, Hguyen, H. (Ed.). Addison-Wesley, Aug. 2007, ch. 39.

[6] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan Primitives for GPU Computing. Graphics Hardware 2007. San Diego, CA, Aug. 2007.

[7] Chatterjee, S., Blelloch, G. E., and Zagha, M. Scan Primitives for Vector Computers. Proceedings of the 1990 Conference on Supercomputing. New York, NY, 1990, pp. 666 – 675.

[8] NVIDIA CUDA SDK. http://developer.nvidia.com/object/cuda.html.

[9] CUDA Data Parallel Primitives Library – CUDPP. http://www.gpgpu.org/developer/cudpp/rel/rel_gems3/html/index.html.

[10] Blelloch, G. E. Scans as Primitive Parallel Operations. Proceeding of the International Conference on Parallel Processing. 1987, pp 355–362.

[11] Horn, D. Stream Reduction Operations for GPGPU Applications. GPU Gems 2, Pharr, M. (Ed.). Addison-Wesley, pp 573–589.

[12] Hensley, J., Scheuermann, T., Coombe, G., Singh, M., Lastra A. Fast Summed-Area Table Generation and its Applications. Computer Graphics Forum 24 (3), pp. 547–555.

[13] Sengupta S., Lefohn A., and Owens, J. A Work-Efficient Step-Efficient Prefix-Sum Algorithm. Proceedings of the Workshop on Edge Computing Using New Commodity Architectures. Chapel Hill, NC, May 2006, pp. 26–27.

[14] Cray-Cyber.org. Cray Y-MP EL. http://www.cray-cyber.org/systems/yel.php.

[15] Hwu, W. and Kirk, D. UIUC ELE 498 AL1: Programming Massively Parallel Processors. http://courses.ece.uiuc.edu/ece498/al1.

[16] Blelloch, G. E., Heroux, M. A., and Zagha, M. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Tech. Rep. CMU-CS-93- 173, School of Computer Science, Carnegie Mellon University, Aug. 1993.

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

[18] Greβ A., Guthe M., Klein R. GPU-based collision detection for deformable parameterized surfaces. Computer Graphics Forum 25, 3 (Sept. 2006), 497-506.

[19] Blelloch, G. E., Siddhartha, C., Marco Z. Solving linear recurrences with loop raking. Journal of Parallel and Distributed Computing, 25(1), pp 91-97, Feb. 2005.