Efficient Parallel Scan Algorithms for GPUs

Shubhabrata Sengupta
University of California, Davis

Mark Harris Michael Garland
NVIDIA Corporation

Abstract

Scan and segmented scan algorithms are crucial building blocks for a great many data-parallel algorithms. Segmented scan and related primitives also provide the necessary support for the flattening transform, which allows for nested data-parallel programs to be compiled into flat data-parallel languages. In this paper, we describe the design of efficient scan and segmented scan parallel primitives in CUDA for execution on GPUs. Our algorithms are designed using a divide-and-conquer approach that builds all scan primitives on top of a set of primitive intra-warp scan routines. We demonstrate that this design methodology results in routines that are simple, highly efficient, and free of irregular access patterns that lead to memory bank conflicts. These algorithms form the basis for current and upcoming releases of the widely used CUDPP library.

1 Introduction

Parallel scan and segmented scan operations are data-parallel primitives whose broad importance is well known. Sequence compaction, radix sort, quicksort, sparse-matrix vector multiplication, and minimium spanning tree construction are only a few of the many algorithms that can be efficiently implemented in terms of scan operations [1, 2]. These operations are the analogs of parallel prefix circuits [13], which have a long history, and have been widely used in collection-oriented languages dating back to APL [12]. They also form the basis for efficiently mapping nested data-parallel languages such as NESL [3] on to flat data-parallel machines.

Because of their fundamental importance, scan operations have been implemented for many parallel systems. The earliest GPU implementations of scan were built using graphics pixel shaders for "non-uniform stream compaction" [11] and summed-area table generation [9]. Sengupta *et al.* [19] adapted Blelloch's work-efficient algorithm [1] to pixel shaders, and Harris *et al.* [8] built on this work to produce the first CUDA-based implementation of scan. Sengupta *et al.* [18] developed the first CUDA implementation of segmented scan by extending the reduce and down-sweep phases of their earlier work, and Dotsenko *et al.* [7] recently adapted the algorithm used by Chatterjee *et al.* [5] for the Cray Y-MP to CUDA.

The model of CUDA programs as a hierarchy of threads, thread blocks, and grids of blocks [15, 16] naturally encourages a hierarchical view of scan algorithms. We can think of a parallel scan kernel as performing some amount of (1) sequential work per thread, (2) parallel combination of results across a thread block, and (3) parallel combination of results between blocks. In this paper, we

NVIDIA Technical Report NVR-2008-003, Dec. 2008.

```
template<class OP, class T>
T scan(T *values, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        values[i] = OP::apply(values[i-1] , values[i]);
}
```

Figure 1. Serial implementation of inclusive scan for generic operator `OP` over values of type `T`.

discuss the design of efficient algorithms for the parallel combination of results across a block of threads. Our block-level algorithms simplify earlier methods [8, 18] considerably, avoid all memory bank conflicts, require no artificial padding, and are substantially more efficient. We retain the per-thread and inter-block methods used by Sengupta *et al.* [18], although expanding the amount of serial work performed per thread is another possible avenue for improving performance [7].

The algorithms outlined in this paper form the core of the scan and segmented scan primitives provided in the most recent release of the widely used CUDPP library [6]. For clarity, we present simplified versions of the kernels in the text, but the full kernels used in our performance profiling are available for download in the library.

2 Parallel Scan Operations

Given an input sequence a and an associative[1] binary operator $\oplus$ with identity I, an *inclusive* scan produces an output sequence $b = \text{scan}\langle\textit{inclusive}\rangle(a, \oplus)$ where $b_i = a_0 \oplus \cdots \oplus a_i$. Similarly, an *exclusive* scan produces an output sequence $b = \text{scan}\langle\textit{exclusive}\rangle(a, \oplus)$ where $b_i = I \oplus \cdots \oplus a_{i-1}$. As a concrete example, consider the input sequence:

```
a = [ 3 1  7  0  4  1  6  3 ]
```

Applying an inclusive scan operation to this array with the usual addition operator produces the result

```
scan<inclusive>(a, +) = [ 3 4 11 11 15 16 22 25 ]
```

and the exclusive scan operation produces the result

```
scan<exclusive>(a, +) = [ 0 3  4 11 11 15 16 22 ]
```

As always, the first element in the result produced by the exclusive scan is the identity element, which in this case is 0.

Implementing scan primitives on a serial processor is trivial, as shown in Figure 1. Note that throughout this paper, we use C++ templates to make scan generic over the operator `OP` and the type of values `T`.

Implementing parallel scan primitives requires somewhat more effort than the trivial serial case. Figure 2 shows a simple implementation of a well-known parallel scan algorithm [4, 10]. This code will scan the binary operator `OP` across an array of $n = 2^k$ values using a single thread block of 2^k

[1] In practice, this requirement is often relaxed to include pseudo-associative operations, such as addition of floating point numbers.

```
template<class OP, class T>
__device__ T scan(T *values)
{
    unsigned int i = threadIdx.x;  // ID of this thread
    unsigned int n = blockDim.x;   // number of threads in block

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset)  t = values[i-offset];
        __syncthreads();

        if(i>=offset)  values[i] = OP::apply(t , values[i]);
        __syncthreads();
    }
}
```

Figure 2. Simple parallel scan of 2^k elements with a single thread block of 2^k threads.

threads. We make these assumptions—that the number of values is a power of 2 and that there is exactly 1 input value per thread—to simplify the presentation of the algorithm.

Analyzing the behavior of this algorithm, we see that it will perform only $\log_2 n$ iterations of the loop, which is optimal. However, this algorithm applies the operator $O(n \log n)$ times, which is asymptotically inefficient compared to the $O(n)$ applications performed by the serial algorithm. It also has practical disadvantages, such as requiring $2 \log_2 n$ barrier synchronizations. In Section 3, we will show how this basic algorithm can be adapted into something that is both asymptotically efficient and very fast in practice.

2.1 Segmented Scan

Segmented scan generalizes the scan primitive by simultaneously performing separate parallel scans on *arbitrary* contiguous partitions ("segments") of the input vector. For example, an inclusive scan of the + operator over a sequence of integer sequences would give the following result:

```
              a = [ [3 1] [7 0  4] [1 6] [3] ]
segscan(a, +) = [ [3 4] [7 7 11] [1 7] [3] ]
```

Segmented scans provide as much parallelism as unsegmented scans, but operate on data-dependent regions. Consequently, they are extremely helpful in mapping irregular computations such as quicksort and sparse matrix-vector multiplication onto regular execution structures, such as CUDA's thread blocks.

Segmented sequences of this kind are typically represented by a combination of (1) a sequence of values and (2) a *segment descriptor* that encodes how the sequence is divided into segments. Of the many possible encodings of the segment descriptor, we focus on using a *head flags* array which stores a 1 for each element that begins a segment and 0 for all others. This representation is convenient for massively parallel machines and all other representations can naturally be converted to this form. The head flags representation for the example sequence above is:

```
a.values = [ 3 1 7 0 4 1 6 3 ]
 a.flags = [ 1 0 1 0 0 1 0 1 ]
```

For simplicity of presentation, we will treat the head flags array as a sequence of 32-bit integers; however, it may in practice be preferable to represent flags as bits packed in words.

Schwartz demonstrated that segmented scan can be implemented in terms of (unsegmented) scan by a transformation of the given operator [17, 1]. Given the operator $\oplus$ we can construct a new operator $\oplus^s$ that operates on flag-value pairs (f_x, x) as follows:

$$(f_x, x) \oplus^s (f_y, y) = (f_x | f_y,\ \text{if } f_y \text{ then } y \text{ else } x \oplus y)$$

Segmented scan can also be implemented directly rather than by operator transformation [5, 18]. In Section 4 we explore both of these implementation strategies.

3 Designing an Efficient Scan

The CUDA programming model requires the programmer to organize parallel kernels into a hierarchy of threads, thread blocks (of at most 512 threads each), and grids of thread blocks. Furthermore, the NVIDIA GPU architecture executes the threads of a block in SIMT (single instruction, multiple thread) groups of 32 called *warps* [14, 15]. While the threads of a warp may follow any execution path, they execute with a shared instruction unit and thus are executing only a single instruction at any instant in time.

This hierarchical grouping of threads naturally encourages a divide-and-conquer approach to designing parallel scan algorithms. To achieve maximum efficiency, we organize our algorithms to match these natural execution granularities. At the lowest level, we design an *intra-warp* primitive to perform a scan across a single warp of threads. We then construct an *intra-block* primitive that composes intra-warp scans together in order to perform a scan across a block of threads. Finally, we combine grids of intra-block scans into a *global* scan of arbitrary length.

In the following sections, we focus primarily on the design of algorithms for intra-block scans. To simplify the discussion, we assume that there is exactly 1 thread per element in the sequence being scanned. Our experience shows that the best efficiency is achieved when each thread initially performs a serial scan of multiple input elements (see Section 6), a finding noted by others as well [7]. The code we present is templated on the input data type and binary operator used, which is assumed to be associative and to possess an identity value. Input arrays to our scan functions (e.g., `ptr` and `hd`) are assumed to be located in fast on-chip shared memory. The code invoking the scan functions is responsible for loading array data from global (off-chip) device memory into (on-chip) shared memory and storing results back.

3.1 Intra-Warp Scan Algorithm

We begin by defining a routine to perform a scan over a warp of 32 threads, shown in Figure 3. It uses precisely the same algorithm as shown in Figure 2, but with a few basic optimizations. First, we take advantage of the synchronous execution of threads in a warp to eliminate the need for barriers. Second, since we know the size of the sequence is fixed at 32, we unroll the loop. We also add the ability to select either an inclusive or exclusive scan via a `ScanKind` template parameter.

For a warp of size w, this algorithm performs $O(w \log w)$ work rather than the optimal $O(w)$ work performed by a work-efficient algorithm [1]. However, since the threads of a warp execute in a SIMT fashion, there is actually no advantage in decreasing work at the expense of increasing

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_warp(volatile T *ptr, const unsigned int idx=threadIdx.x)
{
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)

    if (lane >=  1)  ptr[idx] = OP::apply(ptr[idx -  1] , ptr[idx]);
    if (lane >=  2)  ptr[idx] = OP::apply(ptr[idx -  2] , ptr[idx]);
    if (lane >=  4)  ptr[idx] = OP::apply(ptr[idx -  4] , ptr[idx]);
    if (lane >=  8)  ptr[idx] = OP::apply(ptr[idx -  8] , ptr[idx]);
    if (lane >= 16)  ptr[idx] = OP::apply(ptr[idx - 16] , ptr[idx]);

    if( Kind==inclusive ) return ptr[idx];
    else                  return (lane>0) ? ptr[idx-1] : OP::identity();
}
```

Figure 3. Scan routine for warp of 32 threads with operator `OP` over values of type `T`. The `Kind` parameter is either `inclusive` or `exclusive`.

the number of steps taken. Each instruction executed by the warp has the same cost, whether executed by a single thread or all threads of the warp. Since the work-efficient reduce/downsweep algorithm [1] performs twice as many steps as the algorithm used here, it leads to measurably lower performance in practice.

3.2 Intra-Block Scan Algorithm

We now construct an algorithm to scan across all the threads of a block using this intra-warp primitive. For simplicity, we assume that the maximum block size is at most the square of the warp width, which is true for the GPUs we target. Given this assumption, the intra-block scan algorithm is quite simple.

1. Scan all warps in parallel using inclusive `scan_warp()`.
2. Record the last partial result from each warp i
3. A single warp performs `scan_warp()` on the partial results from Step 2.
4. Each thread of warp i accumulates partial results from Step 3 into its output element from Step 1.

This organization of the algorithm is only possible because of our assumption that the scan operator is associative. The CUDA implementation of this algorithm is shown in Figure 4. The individual steps are labeled and correspond to the algorithm outline.

3.3 Global Scan Algorithm

The `scan_block()` routine performs a scan of fixed size, corresponding to the size of the thread blocks. We use this routine to construct a "global" scan routine for sequences of any length as follows.

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_block(volatile T *ptr, const unsigned int idx=threadIdx.x)
{
    const unsigned int lane   = idx & 31;
    const unsigned int warpid = idx >> 5;

    // Step 1: Intra-warp scan in each warp
    T val = scan_warp<OP,Kind>(ptr, idx);
    __syncthreads();

    // Step 2: Collect per-warp partial results
    if( lane==31 )  ptr[warpid] = ptr[idx];
    __syncthreads();

    // Step 3: Use 1st warp to scan per-warp results
   if( warpid==0 ) scan_warp<OP,inclusive>(ptr, idx);
   __syncthreads();

    // Step 4: Accumulate results from Steps 1 and 3
   if (warpid > 0) val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    // Step 5: Write and return the final result
    ptr[idx] = val;
    __syncthreads();

    return val;
}
```

Figure 4. Intra-block scan routine composed from `scan_warp()` primitives.

1. Scan all blocks in parallel using `scan_block()`.

2. Store the partial result from each block i to `block_results[`i`]`.

3. Perform a scan of `block_results`.

4. Each thread of block i adds element i from Step 3 to its output element from Step 1.

Because they require global synchronization, Steps 1 & 2, 3, and 4 require 3 separate CUDA kernel invocations. Indeed, Step 3 may require repeated application of the global scan algorithm if the number of blocks in Step 1 is greater than the block size.

Aside from the decomposition into kernels, the structure of this global algorithm is strikingly similar to the intra-block algorithm. Indeed, they are nearly identical except for Step 3, where the fixed width of blocks guarantees that the intra-block routine can scan per-warp partial results using a single warp, while the variable block count necessary in the global scan does not provide an analogous guarantee.

4 Efficient Segmented Scan

To implement efficient segmented scan routines, we follow the same design strategy already outlined in Section 3 for scan. We begin by defining an intra-warp primitive, from which we can build an intra-block primitive, and ultimately a global segmented scan algorithm. The implementations are also quite similar, with the added complications of dealing with arrays of head flags.

4.1 Operator Transformation

As described in Section 2.1, segmented scan can be implemented by transforming the operator $\oplus$ into a segmented operator $\oplus^s$ that operates on flag–value pairs [17]. This leads to a particularly simple strategy of defining a `segmented<>` template such that `scan<segmented<OP>>` applied to an array of flag–value pairs accomplishes the desired segmented scan. Sample code for such a transformer is shown in Figure 5. This trivially converts the inclusive `scan_warp()` and `scan_block()` routines given in Section 3 into segmented scans. Achieving a correct exclusive segmented scan via operator transformation requires additional changes to the inclusive/exclusive logic in these routines.

Although a reasonable approach, one downside of relying purely on operator transformation is that it alters the external interface of the scan routines. It accepts a sequence of flag–value pairs rather than corresponding sequences of values and flags. We can restore our desired interface, and simultaneously accommodate correct handling of exclusive scans, by explicitly expanding `scan_warp<segmented<OP>>()` into the `segscan_warp()` routine shown in Figure 6. The structure of this procedure is exactly the same as the one shown in Figure 3 except that (1) it does roughly twice as many operations and (2) requires slightly different logic for determining the final inclusive/exclusive result.

4.2 Direct Intra-Warp Segmented Scan

We have also explored an alternative technique for adapting our basic `scan_warp` procedure into an intra-warp segmented scan. This routine, which is shown in Figure 7, operates by augmenting the conditionals used in the indexing of the successive steps of the algorithm. Each thread computes the index of the head of its segment, or 0 if the head is not within its warp. This is the *minimum*

```
template<class OP>
struct segmented
{
    template<class T>
    static __host__ __device__
    inline T apply(const T a, const T b)
    {
        T c;
        c.flag  = b.flag | a.flag;
        c.value = b.flag ? b.value : OP::apply(a.value, b.value);
        return c;
    }
};
```

Figure 5. Code for transforming operator `OP` on values of type `T` into an operator `segmented<OP>` on flag–value pairs.

index of the segment, and is recorded in the variable `mindex`. We compute `mindex` by writing the index of each segment head to the `hd` array and propagating it within the warp to other elements of its segment via a max-scan operation. We take advantage of the unpacked format of the head flags and use them as temporary scratch space.

We use the minimum segment indices to guarantee that elements from different segments are never accumulated. The unsegmented routine shown in Figure 3 is essentially just the special case where `mindex=0`. An example of the resulting data movement for a warp of size 8 is illustrated in Figure 8.

4.3 Block and Global Segmented Scan Algorithms

Either of the intra-warp segmented scan routines we have just outlined can be used to build an intra-block segmented scan. The methodology is essentially identical to our construction of the intra-block scan routine in Section 3.2, with two additional complications. First, when writing the partial result produced by the last thread of each warp in Step 2, we also write an aggregate segment flag for the entire warp. This flag indicates whether there is a segment boundary within the warp, and is simply the (implicit) or-reduction of the flags of the warp. Second, we accumulate the per-warp offsets in Step 4 only to elements of the first segment of each warp. This process is illustrated in Figure 9.

The full CUDA implementation of this algorithm is shown in Figure 10. We first record if the warp starts with a new segment, because the flags array is converted to `mindex`-form by the first `segscan_warp()` call. Step 2b determines whether any flag in a warp was set. This step also determines if the thread belongs to the first segment in its warp by checking if (1) the first element of the warp is not the first element of a segment (the warp is *open*) and (2) the index of the head of the segment is 0. The remaining steps compute warp offsets using a segmented scan of per-warp partial results and accumulates them to the per-thread results computed in Step 1.

The global segmented scan algorithm can be built in the same way. We observe that another way to check if a thread belongs to the first segment of a warp (or block) is to do a min-reduction of `hd`. This gives the index of the first element of the second segment in each warp. Each thread

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_warp(volatile T *ptr, volatile flag_type *hd,
                          const unsigned int idx = threadIdx.x)
{
   const unsigned int lane = idx & 31;

   if (lane >=  1) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 1] , ptr[idx]);
       hd[idx]  = hd[idx -  1] | hd[idx];
   }
   if (lane >=  2) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 2] , ptr[idx]);
       hd[idx]  = hd[idx -  2] | hd[idx];
   }
   if (lane >=  4) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 4] , ptr[idx]);
       hd[idx]  = hd[idx -  4] | hd[idx];
   }
   if (lane >=  8) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 8] , ptr[idx]);
       hd[idx]  = hd[idx -  8] | hd[idx];
   }
   if (lane >= 16) {
       ptr[idx] = hd[idx] ? ptr[idx] : OP::apply(ptr[idx - 16] , ptr[idx]);
       hd[idx]  = hd[idx - 16] | hd[idx];
   }

   if( Kind==inclusive )  return ptr[idx];
   else  return (lane>0 && !flag) ? ptr[idx-1] : OP::identity();
}
```

Figure 6. Intra-warp segmented scan derived by expanding `scan_warp<segmented<OP>>`.

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_warp(volatile T *ptr, volatile flag_type *hd,
                          const unsigned int idx = threadIdx.x)
{
    const unsigned int lane = idx & 31;

    // Step 1: Convert head flags to minimum-index form
    if( hd[idx] ) hd[idx] = lane;
    flag_type mindex = scan_warp<op_max, inclusive>(hd);

    // Step 2: Perform segmented scan across warp of size 32
    if( lane >= mindex + 1 )  ptr[idx] = OP::apply(ptr[idx - 1] , ptr[idx]);
    if( lane >= mindex + 2 )  ptr[idx] = OP::apply(ptr[idx - 2] , ptr[idx]);
    if( lane >= mindex + 4 )  ptr[idx] = OP::apply(ptr[idx - 4] , ptr[idx]);
    if( lane >= mindex + 8 )  ptr[idx] = OP::apply(ptr[idx - 8] , ptr[idx]);
    if( lane >= mindex +16 )  ptr[idx] = OP::apply(ptr[idx -16] , ptr[idx]);

    // Step 3: Return correct value for inclusive/exclusive kinds
    if( Kind==inclusive )  return ptr[idx];
    else  return (lane>0 && mindex!=lane) ? ptr[idx-1] : OP::identity();
}
```

Figure 7. Intra-warp segmented scan using conditional indexing.

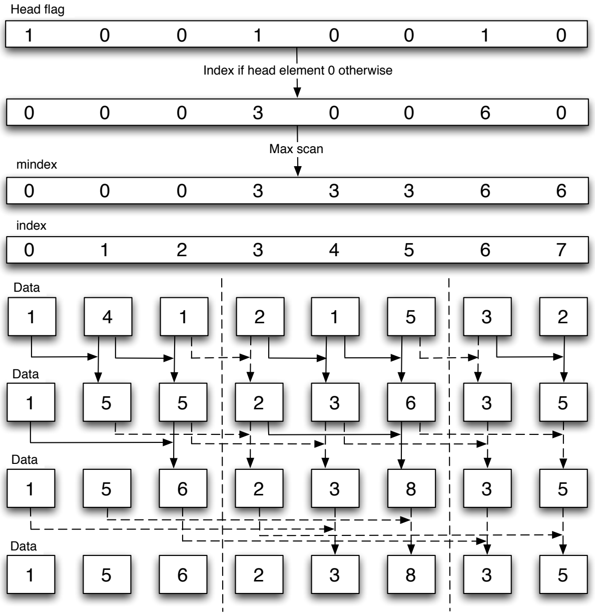

Figure 8. Data movement in intra-warp segmented scan code shown in Figure 7 for threads 0–7 of an 8-thread warp. Data movement in the unsegmented case (dotted arrows) crossing segment boundaries (vertical dashed lines) are not allowed.

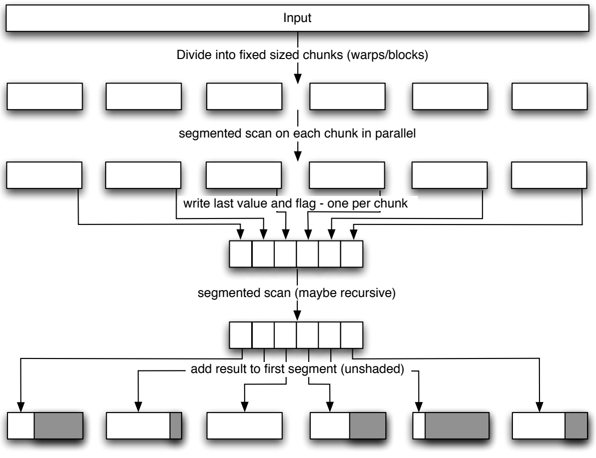
Input
Divide into fixed sized chunks (warps/blocks)
segmented scan on each chunk in parallel
write last value and flag - one per chunk
segmented scan (maybe recursive)
add result to first segment (unshaded)

Figure 9. Constructing block/global segmented scans from warp/block segmented scans.

then checks if its thread index is less than this index before adding the result in Step 4. This is a typical time vs. space tradeoff. The method used in Figure 10 must carry one value per thread but no reduction is necessary; this alternative must carry only one value per warp but requires a reduction. We use the former when propagating data between warps because shared memory loads and stores are cheap. However, when we are doing a global segmented scan, Steps 1 and 3 happen in different kernel invocations, so data must be written to global memory which is much slower. Therefore for global scans we do a min-reduction and write only one index per block instead of one per thread.

5 Algorithmic Complexity

Given the hierarchy of blocks and warps described above, we can calculate the algorithmic complexity to scan n elements. Let B and w represent the block and warp size in threads, respectively. We assume that each block scans $O(B)$ elements (i.e., $O(1)$ elements per thread). To scan n elements we use n/B blocks. Let S and W denote step and work complexity, respectively. The *work complexity* of an algorithm is defined as the total number of operations the algorithm performs. Multiple operations may be performed in parallel in each step. The *step complexity* of an algorithm is the number of steps required to complete the algorithm given an infinite number of threads/processors; this is essentially equivalent to the critical path length through the data dependency graph of the computation. A subscript of n, b or w indicates complexities of for the whole array, a block, or a warp, respectively.

As already discussed, our `scan_warp()` routine has step complexity $S_w = O(\log_2 w)$ and work complexity $W_w = O(w \log_2 w)$. For a block containing B/w warps, we arrive at the following step and work complexity for a block scan.

$$S_b = O(S_w \log_w B) = O\left((\log_2 w)\left(\frac{\log_2 B}{\log_2 w}\right)\right) = O(\log_2 B) \qquad (1)$$

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_block(volatile T *ptr, volatile flag_type *hd,
                           const unsigned int idx = threadIdx.x)
{
    unsigned int warpid     = idx >> 5;
    unsigned int warp_first = warpid << 5;
    unsigned int warp_last  = warp_first + 31;

    // Step 1a:
    // Before overwriting the input head flags, record whether
    // this warp begins with an "open" segment.
    bool warp_is_open = (hd[warp_first] == 0);
    __syncthreads();

    // Step 1b:
    // Intra-warp segmented scan in each warp.
    T val = segscan_warp<OP,Kind>(ptr, hd, idx);

    // Step 2a:
    // Since ptr[] contains *inclusive* results, irrespective of Kind,
    // the last value is the correct partial result.
    T warp_total = ptr[warp_last];

    // Step 2b:
    // warp_flag is the OR-reduction of the flags in a warp and is
    // computed indirectly from the mindex values in hd[].
    // will_accumulate indicates that a thread will only accumulate a
    // partial result in Step 4 if there is no segment boundary to its left.
    flag_type warp_flag = hd[warp_last]!=0 || !warp_is_open;
    bool will_accumulate = warp_is_open && hd[idx]==0;

    __syncthreads();

    // Step 2c: The last thread in each warp writes partial results
    if( idx == warp_last )
    {
        ptr[warpid] = warp_total;
        hd[warpid]  = warp_flag;
    }
    __syncthreads();

    // Step 3: One warp scans the per-warp results
    if( warpid == 0 )
        segscan_warp<OP,inclusive>(ptr, hd, idx);

    __syncthreads();

    // Step 4: Accumulate results from Step 3, as appropriate.
    if( warpid != 0 && will_accumulate )
        val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    ptr[idx] = val;
    __syncthreads();

    return val;
}
```

Figure 10. Intra-block segmented scan routine built using intra-warp segmented scans.

$$W_b = O\left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil W_w\right) = w \log_2 w \left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil\right) = O\left(B \log_2 w\right) \tag{2}$$

The same pattern extends to the array (global) level. The step and work complexity for an array comprising an arbitrary number of blocks, n/B is given by the following expressions.

$$S_n = O(S_b \log_B n) = O((\log_2 w)(\log_B n)) = O(\log_2 n) \tag{3}$$

$$W_n = O\left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil W_b\right) = w \log_2 w \left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil \sum_{j=1}^{\log_w B} \left\lceil \frac{B}{w^j} \right\rceil\right) = O\left(n \log_2 w\right) \tag{4}$$

For any given machine, it is safe to assume that the warp size w is in fact some constant number. Under this assumption, the step complexity of our algorithm is $S_n = O(\log n)$ and the work complexity is $W_n = O(n)$, both of which are asymptotically optimal.

6 Optimizations in CUDPP

The code given in Sections 3 and 4 illustrate the core parallel scan algorithms we use. However, to achieve peak performance for scan kernels, CUDPP combines these basic algorithms with an orthogonal set of further optimizations.

The biggest efficiency gain comes from optimizing the amount of work performed by each thread. We find that processing one element per thread does not generate enough computation to hide the I/O latency to off-chip memory, and so we assign eight input elements to each thread. In our implementations this is handled when data is loaded from global device memory into shared memory. Each thread reads two groups of four elements from global memory and scans both groups of four sequentially. The rightmost result in each group of four (i.e., the reduction of the four inputs) is fed as input to a routine very similar to the block level routines shown in Figure 4 and Figure 10. The key difference is that the block level routines are slightly modified to handle two inputs per thread instead of one as shown here. When the block level routine terminates, the result is accumulated back to the groups of four elements which were scanned serially.

The next most important set of optimizations focuses on minimizing the number of registers used. The GPU architecture relies on multithreading to hide memory access latency, and the number of threads that can be co-resident at one time is often limited by their register requirements. Therefore, it is important to maximize the number of available co-resident threads by minimizing register usage. Optmizing register usage is particularly important for segmented scan since many more registers are needed to store and manipulate head flags. The CUDPP code uses a number of low-level code optimizations designed to limit register requirements, including packing multiple head flags into the bits of registers after they are loaded from off-chip memory.

7 Performance Analysis

In this section, we analyze the performance of the scan and segmented scan routines that we have described and compare them to some alternative approaches. All running times were collected on an NVIDIA GeForce GTX 280 GPU with 30 Streaming Multiprocessors (SMs). These measurements do not include any transfers of data between CPU and GPU memory, under the assumption that scan and segmented scan will be used as building blocks in more complicated applications on GPUs.

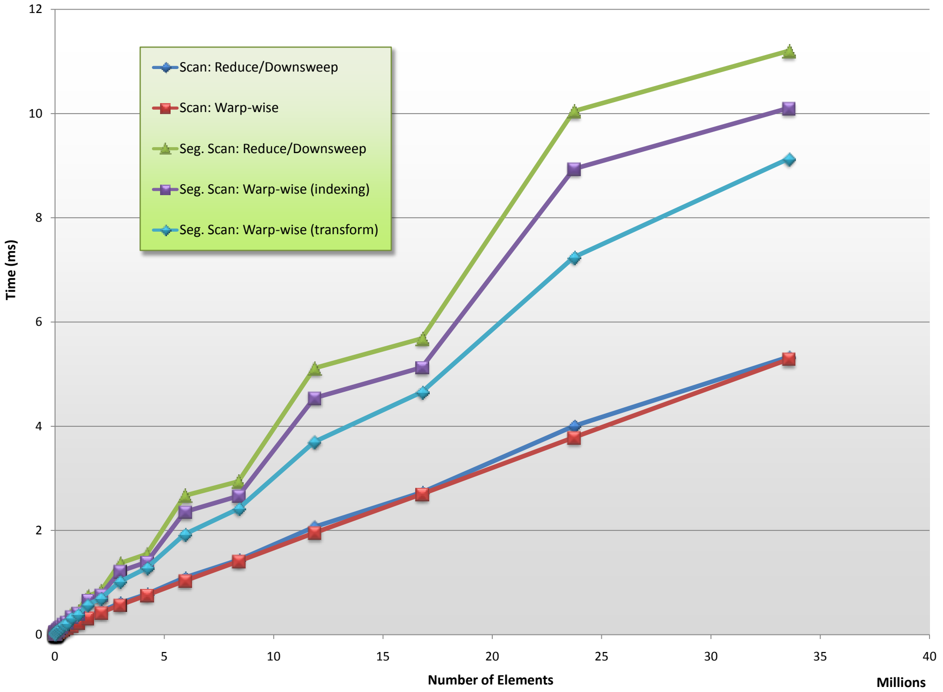

Figure 11. Scan and Segmented Scan performance.

Our first test consists of running both scan and segmented scan routines over sequences of varying length using the CUDPP test apparatus. These tests scan the addition operator over sequences of floating point values.

Figure 11 compares two scan implementations: our warp-wise scan and the reduce/downsweep algorithm used by Sengupta *et al.* [18]. Our warp-wise approach is 5 to 20% faster, improving most on scans of non-power-of-two arrays. Figure 11 compares the reduce/downsweep segmented scan [18] with both of our warp-wise segmented scan kernels: one based on conditional indexing (Fig. 7) and one based on operator transformation (Fig. 6). We see the same trend as in the unsegmented case. Our direct warp-based algorithm using conditional indexing is up to 29% faster than the reduce/downsweep algorithm. The kernel derived from operator transformation improves on it by a further 6 to 10%. Compared to the results reported by Sengupta *et al.* [18] for sequences of 1,048,576 elements running on an older NVIDIA GeForce 8800 GTX GPU, our scan implementation is 2.8 times faster and our segmented scan is 4.2 times faster on the same hardware.

Multiple factors contribute to the performance increase in scan and segmented scan. First, using warp-wise execution minimizes the need for barrier synchronization, because most thread communication is within warps. In contrast, the reduce/downsweep algorithm requires barrier synchronizations between each step in both the reduce and downsweep stages. Second, we halve the number of parallel steps required, as compared to the reduce/downsweep algorithm. This comes at the “expense” of additional work, which is actually not a cost at all due to the SIMT execution of warps. The intra-warp step complexity is in fact optimal. Third, our segmented scan based on operator transformation interleaves scans of flags and data. Like software pipelining, this increases the distance between dependent instructions. Consequently, the performance of this kernel is higher than the indexing kernel, which performs the scan of the flags before the scan of the values. Finally,

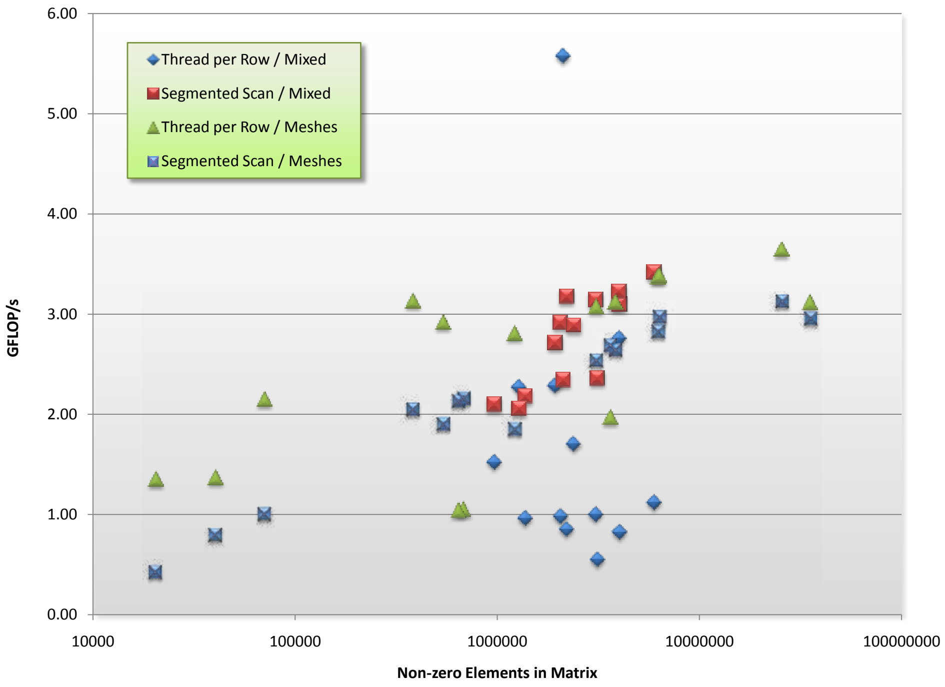

Figure 12. Comparative performance for sparse matrix-vector product kernels using segmented scan vs. assigning 1 thread per row.

while our reduce/downsweep implementations expend much effort to avoid shared memory bank conflicts [8], the intra-warp scan algorithm is inherently conflict-free.

The main efficiency advantage of segmented scan is that its performance is largely invariant to the way in which the sequence is decomposed into subsequences. Thus, it implicitly load-balances work across potentially quite unbalanced subsequences. An example case where this phenomenon is important is in sparse-matrix vector multiplication, where the sparse matrix is represented in a CSR (compressed sparse row) format. One strategy for implementing this computation in CUDA is to assign 1 thread to process each row, examples of which are given by Nickolls *et al.* [15]. Alternatively, sparse-matrix vector multiplication can be implemented with a combined gather and segmented scan [3]. Figure 12 demonstrates the performance of two such kernels on both a corpus of mixed matrices and triangle mesh Laplacian matrices.

The performance of the kernel which simply assigns 1 thread per row has a much higher variance (1.39) as opposed to the kernel using segmented scan (0.55). Furthermore, the segmented scan achieves a higher median performance of 2.59 GFLOP/s as compared to a median of 2.06 GFLOP/s for the thread-per-row kernel. For the mesh matrices, which are all reasonably uniform, the performance rates of the segmented scan are tightly clustered—with the exception of 3 small matrices where overhead is a large cost. Even on the more varied mixed corpus, the performance rates are similar. In contrast, while the thread-per-row kernel performs quite well on certain matrices with uniformly short rows, it suffers considerably on matrices with many very long rows or imbalanced row lengths.

Finally, Figure 13 illustrates the scaling of our segmented scan algorithm on sequences of various sizes. We use the running time on 3 SMs of a GeForce GTX 280 as a base line and show the

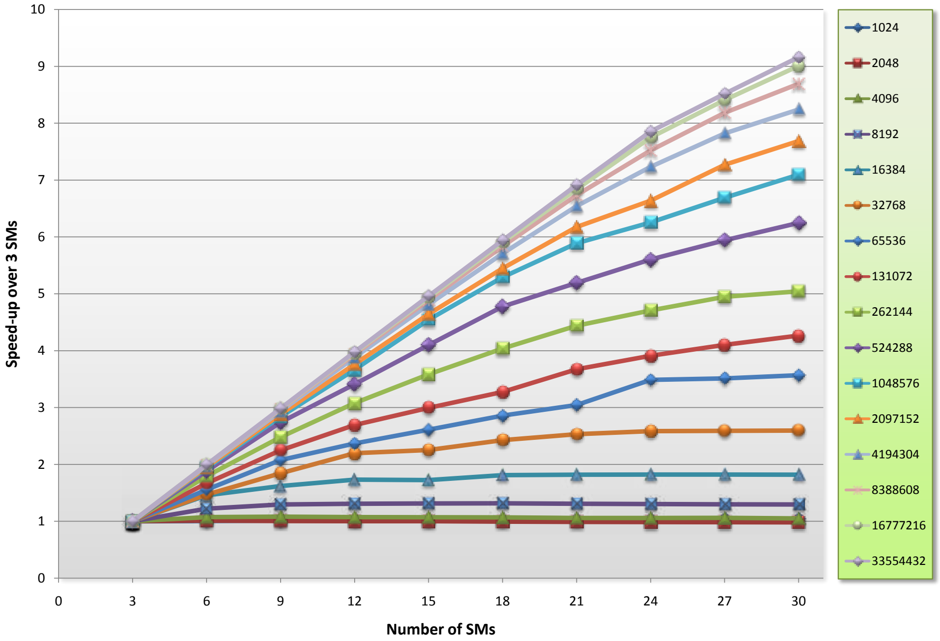

Figure 13. Parallel scaling of segmented scan on sequences of varying length.

speed-up achieved over this base line for 6–30 SMs. Small sequences show relatively little scaling, as they are small enough to be processed efficiently by a small number of SMs. For sequence sizes above 512K elements, we see strong linear scaling. Scaling results for the scan algorithm are similar. This demonstrates the scalability of both the GPU architecture itself and our algorithmic design.

8 Conclusions

The modern manycore GPU is a massively parallel processor and the CUDA programming model provides a straightforward way of writing scalable parallel programs to execute on the GPU. Because of its deeply multithreaded design, a program must expose substantial amounts of fine-grained parallelism to efficiently utilize the GPU. Data-parallel techniques provide a convenient way of expressing such parallelism. Furthermore, the GPU is designed to deliver maximum performance for regular execution paths—via its SIMT architecture—and regular data access patterns—via memory coalescing—and data-parallel algorithms generally fit these expectations quite well.

We have described the design of efficient scan and segmented scan routines, which are essential primitives in a broad range of data-parallel algorithms. By tailoring our algorithms to the natural granularities of the machine and minimizing synchronization, we have produced one of the fastest scan and segmented scan algorithms yet designed for the GPU. The scan and segmented scan routines based on the algorithms described in this paper are available as part of the CUDA Data Parallel Primitives (CUDPP) library, which can be obtained from `http://www.gpgpu.org/developer/cudpp/`.

References

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

[2] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, 1989.

[3] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[4] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[5] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 Conference on Supercomputing*, pages 666–675, 1990.

[6] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, 2008.

[7] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proc. 22nd Annual International Conference on Supercomputing*, pages 205–213. ACM, June 2008.

[8] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, Aug. 2007.

[9] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, Sept. 2005.

[10] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[11] D. Horn. Stream reduction operations for GPGPU applications. In M. Pharr, editor, *GPU Gems 2*, chapter 36, pages 573–589. Addison Wesley, Mar. 2005.

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

[13] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.

[16] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, June 2008. Version 2.0.

[17] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct. 1980.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.

[19] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.