Scan Primitives for GPU Computing

Shubhabrata Sengupta, Mark Harris*, Yao Zhang, and John D. Owens

University of California, Davis   *NVIDIA Corporation

Abstract

The scan primitives are powerful, general-purpose data-parallel primitives that are building blocks for a broad range of applications. We describe GPU implementations of these primitives, specifically an efficient formulation and implementation of segmented scan, *on NVIDIA GPUs using the CUDA API. Using the scan primitives, we show novel GPU implementations of quicksort and sparse matrix-vector multiply, and analyze the performance of the scan primitives, several sort algorithms that use the scan primitives, and a graphical shallow-water fluid simulation using the scan framework for a tridiagonal matrix solver.*

1. Introduction and Motivation

By the end of 2007, the most advanced processors will surpass one billion transistors on a single chip. This wealth of computational resources has yielded GPUs that can sustain hundreds of GFLOPS on not just graphics applications but also a wide variety of computationally demanding general-purpose problems.

The primary reason that GPUs deliver such high performance is that the GPU is a highly parallel machine. NVIDIA's latest flagship GPU, for instance, boasts 128 processors. GPUs keep these processors busy by juggling thousands of parallel computational threads. Graphics workloads are perfectly suited to delivering large amounts of fine-grained parallel work. The programmable parts of the graphics pipeline operate on primitives (vertices and fragments), with hundreds of thousands to millions of each type of primitive in a typical frame. These primitive programs spawn a thread for each primitive to keep the parallel processors full.

It is instructive to look at the graphics programming model as the GPU becomes more general purpose. Let us consider fragment processing as a representative example. In both the OpenGL and DirectX APIs, fragment processing is completely data-parallel. The fragment processors iterate over each input fragment, producing a fixed number of outputs per fragment that depend only on the incoming fragment (Lefohn et al. call this access pattern a "single-access iterator" [LKS*06]). It is this explicit data parallelism that enables the effective use of so many parallel processors in recent GPUs.

This explicit parallelism, in turn, is well suited for a *stream programming model* in which a single program (a *kernel*) operates in parallel on each input element, producing one output element for each input element. The programmable fragment and vertex parts of the graphics pipeline precisely match this strict stream programming model. The stream model extends nicely to problems where each output element is a function not just of a single input but of a small, bounded neighborhood of inputs (a "neighborhood-access iterator"). Most general-purpose applications that have been mapped efficiently to GPUs fit nicely into this more general stream programming model (for instance, particle systems, image processing, grid-based fluid simulations, and dense matrix algebra).

1.1. More Complex Operations are Needed

Consider a fragment program that operates on n fragments. With a single-access iterator, each output fragment must access a single input fragment; with a neighborhood-access iterator, each output fragment must access a small bounded number of input fragments. The total number of accesses necessary to compute all fragments is $O(n)$.

Many interesting problems, however, have more complex access requirements than we can support with a single- or neighborhood-access iterator. It is these problems that we address in this paper. A common algorithmic pattern that arises in many parallel applications with complex access requirements is the *prefix-sum* algorithm. The input to prefix-sum is an array of values. The output is an equally sized array in which each element is the sum of all values that preceded it in the input array:

```
in: 3  1  7  0  4  1  6  3
out: 0  3  4  11  11  14  16  22
```

How do we compute the value of the last element in the output (22)? That value is a function of *every* value in the input stream. With a serial processor, such a computation is trivial, but with a parallel processor, it is more difficult. The naive way to compute each output in parallel is for every element in the output stream to sum all preceding values from the input stream. This approach requires $O(n^2)$ total memory accesses and addition operations. This cost is too high.

1.2. Scan: An Efficient Parallel Primitive

We are interested in finding efficient solutions to parallel problems in which *each output requires global knowledge of the inputs*. We attack these problems using a family of algorithms called the *scan* primitives. Our scan implementation has a serial work complexity of $O(n)$. While the standard scan primitive was introduced to the GPU by Horn [Hor05], in this paper we introduce the *segmented scan* primitive to the GPU, and present new approaches to implementing several classic applications using the scan primitives.

2. Primitives

We chose NVIDIA's CUDA GPU Computing environment [NVI07] for our implementation. CUDA provides a direct, general-purpose C language interface to the programmable processors on NVIDIA's 8-series GPUs, eliminating much of the complexity of writing GPGPU applications using graphics APIs such as OpenGL. Furthermore, CUDA exposes some important new hardware features that have large benefits to the performance of data-parallel computations:

General Load-Store Memory Architecture CUDA allows arbitrary gather and scatter memory access from GPU programs.
On-chip Shared Memory Each multiprocessor on the GPU contains a fast on-chip memory (16 kB on NVIDIA 8-series GPUs). All threads running on a multiprocessor can load and store data from this memory.
Thread Synchronization A barrier instruction is provided to synchronize between all threads active on a GPU multiprocessor. Together with shared memory, this feature allows threads to cooperatively compute results.

NVIDIA's 8-series GPUs feature multiple physical multiprocessors, each with a shared memory and multiple scalar processors (for example, the NVIDIA GeForce 8800 GTX has 16 multiprocessors with eight processors each). CUDA structures GPU programs into parallel *thread blocks* of up to 512 SIMD-parallel threads. Programmers specify the number of thread blocks and threads per block, and the hardware and drivers map thread blocks to parallel multiprocessors on the GPU. Within a thread block, threads can communicate through shared memory and cooperate by combining shared memory with thread synchronization.

Efficient CUDA programs exploit both thread parallelism within a thread block and coarser block parallelism across thread blocks. Because only threads within the same block can cooperate via shared memory and thread synchronization, programmers must partition computation into multiple blocks. While this adds complexity to the programming model compared to using a single partition (such as with pixel shaders in the graphics API), the potential performance benefits are large. For instance, we previously compared our optimized OpenGL unsegmented scan implementation against our CUDA implementation on the same GPU and found a 7× speedup for large scans [HSO07].

2.1. Scan

Scan, first proposed as part of APL [Ive62], was popularized by Blelloch as a fundamental primitive on the Connection Machine [Ble90]. On the GPU, scan was first used by Horn for "non-uniform stream compaction" [Hor05] as part of a collision-detection application. Hensley et al. improved Horn's implementation in their summed-area-table work [HSC*05], but the serial work complexity of both Horn and Hensley's algorithms was $O(n\log n)$. The next year Sengupta et al. and Greß et al. demonstrated the first $O(n)$ GPU-based scan implementations [SLO06, GGK06].

The inputs to a scan operation are a vector of data elements and an associative binary function $\oplus$ with an identity element $i^\dagger$. If the input is $[a_0, a_1, a_2, a_3, \ldots]$, an *exclusive* scan produces the output $[i, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]$, while an *inclusive* scan produces the output $[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, a_0 \oplus a_1 \oplus a_2 \oplus a_3, \ldots]$. Note that the scan output requires global knowledge of all inputs, as we discussed in Section 1; the output at the last element is a function of all inputs. In this paper, we implement an exclusive sum-scan as our fundamental primitive and generate inclusive scan by adding the input vector to the exclusive output vector. We also support backward scans by reversing the input elements at the beginning of the scan (typically when they are read from GPU main memory).

We briefly outline the details of the $O(n)$ unsegmented CUDA scan implementation we use in this paper, which is more fully described in our previous work [HSO07]. The implementation logically follows the work-efficient formulation of Blelloch [Ble90] and the GPU implementation of Sengupta et al. [SLO06], but is adapted for efficiency on CUDA. Our work-efficient scan of *n* elements requires two passes over the array, called *reduce* and *down-sweep*, shown in Algorithm 1 and 2, respectively, and depicted in Figure 1. Each of these two passes requires $\log n$ parallel steps. The amount of work is cut in half at each step, resulting in an overall work complexity of $O(n)$.

[†] Typical binary functions are plus [with identity 0], min, max, logical and, and logical or.
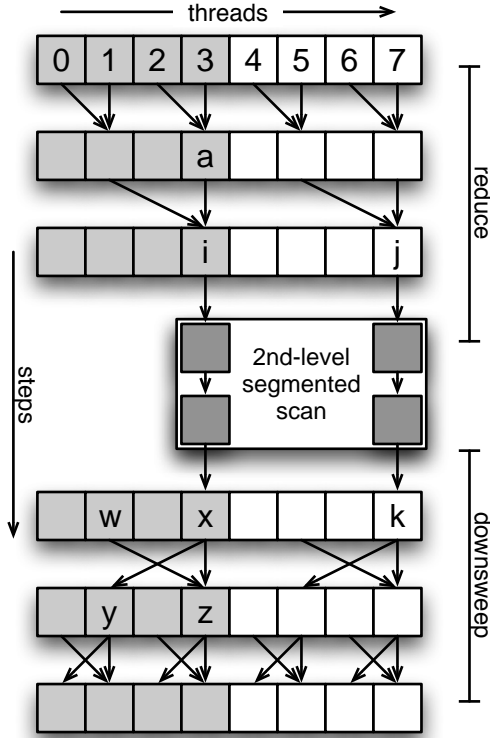
Figure 1: *Multi-block segmented scan communication. Cells with the same shading belong to the same block. The example above processes 4 threads per block and requires three steps: 2 blocks running in parallel for the reduce step, 1 block to combine the results from the reduce step and prepare for the downsweep, and 2 blocks running in parallel in the downsweep. Letters refer to the discussion in Section 2.2.1.*

Each thread processes two elements; if the number of elements exceeds the maximum number that a single thread block can process, the array is divided across multiple thread blocks and the partial sum tree results are used as input to a second-level recursive scan. Each element of the output of this scan is then added to all elements of the corresponding block in the first-level scan. This recursion continues as necessary for scans of very large arrays.

1: **for** $d = 0$ to $\log_2 n - 1$ **do**
2:     **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
3:         $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$

Algorithm 1: The reduce (up-sweep) phase of a work-efficient parallel unsegmented scan algorithm.

1: $x[n-1] \leftarrow 0$
2: **for** $d = \log_2 n - 1$ down to 0 **do**
3:     **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
4:         $t \leftarrow x[k + 2^d - 1]$
5:         $x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1]$
6:         $x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1]$

Algorithm 2: The down-sweep phase of a work-efficient parallel unsegmented scan algorithm.

2.2. Segmented Scan

The major contribution of this paper is the introduction of the *segmented scan* primitive [Sch80] to the GPU. Segmented scan generalizes the scan primitive by allowing parallel scans on *arbitrary* partitions ("segments") of the input vector. Segments are demarcated by flags, where a set flag marks the first element of a segment. Our segmented scan implementation has the same $O(n)$ complexity as scan and is only three times slower than scan. Segmented scan is useful as a building block for a broad variety of applications [Ble90] that have not previously been efficiently implemented on GPUs.

2.2.1. Work-efficient segmented scan algorithm

Our work-efficient segmented scan implementation follows the high-level structure of our work-efficient implementation of scan [HSO07], shown in Figure 1. Squares that have the same color (light gray or white) represent threads that belong to the same block. The arrows show data movement. Two arrows entering the same square signify a binary operation, which can be a copy, a binary scan operation, or a logical OR. In the discussion below, we use sum as the binary scan operation, but the algorithm works for any other binary associative operator.

Our contribution is to extend the reduce and down-sweep phases of the unsegmented scan algorithm to efficiently implement segmented scan on the GPU. Though Schwartz first presented the concept of segmented scans [Sch80], he did not describe how segmented scan can be implemented using scan's balanced-tree approach. Chatterjee et al.'s implementation of segmented scan [CBZ90] was closely tied to the Cray-MP architecture. They used the wide vector registers to carve up large input arrays into smaller chunks. Within each block the segmented scan ran serially. The flags were stored in vector mask registers of the Cray-MP which made accessing them quite efficient using the merge operation.

We also show how to factor the algorithm into chunks, which is necessary when the input vector is too long to be processed in shared memory by a single thread block. The simple parallelization structure of unsegmented scan is not directly applicable to segmented scan.

Just as in scan, the segmented reduce phase traverses a binary tree with n leaves and $d = \log_2 n$ levels with 2^d nodes each. Reduce traverses up the tree from the leaves to the root

computing partial-sum results at each internal node of the tree. For example, in Figure 1, a is the sum of the values in threads 2 and 3. Segmented scan must compute intermediate results for both the data and the head flags. The partial OR flag is simply the logical OR of two input head flags. The partial data sums are computed just as in unsegmented scan unless the right parent's head flag is set, in which case the right parent's data element is taken unmodified. Pseudocode for the reduce phase is shown in Algorithm 3.

1: **for** $d = 1$ to $\log_2 n - 1$ **do**
2:    **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
3:       **if** $f[k + 2^{d+1} - 1]$ is not set **then**
4:          $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
5:          $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^d - 1] \mid f[k + 2^{d+1} - 1]$

Algorithm 3: The reduce (up-sweep) phase of a segmented scan algorithm. x denotes the partial sums and f denotes the partial OR flags.

As in scan, the down-sweep phase traverses back down the tree from the root using the partial sums from the reduce phase. The key difference from the unsegmented scan algorithm is that the right child (z in Figure 1) is not always set to the sum of its parent's value (x) and the former value of its left sibling (w). The right child (z) of x is set to 0 (the identity element) if the flag in the position to the right of x is set in the input flag vector. Otherwise, if the partial OR stored in the same position in the tree is set, the right child (z) is set to the original value of its left sibling (w). If neither flag is set, the right child (z) receives the sum of its parent (x) and the original value of its left sibling (w). The pseudocode for the down-sweep phase is shown in Algorithm 4.

1: $x[n - 1] \leftarrow 0$
2: **for** $d = \log_2 n - 1$ down to 0 **do**
3:    **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
4:       $t \leftarrow x[k + 2^d - 1]$
5:       $x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1]$
6:       **if** $f_i[k + 2^d]$ is set **then**
7:          $x[k + 2^{d+1} - 1] \leftarrow 0$
8:       **else if** $f[k + 2^d - 1]$ is set **then**
9:          $x[k + 2^{d+1} - 1] \leftarrow t$
10:      **else**
11:         $x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1]$
12:      Unset flag $f[k + 2^d - 1]$

Algorithm 4: The down-sweep phase of a segmented scan algorithm.

2.2.2. Segmented scan implementation

Besides the algorithmic difference, our segmented scan implementation differs from our scan implementation [HSO07] in two major ways: the representation and storage of flags and the merging of results for multi-block segmented scans.

Head flag representation and API In our implementation, segments are denoted by a head flag vector that has the same length as the input vector. If an element in the input vector is the first element of a segment, the corresponding entry in the head flag is set to 1. All other entries in the head flag vector are set to 0. Even though head flags appear on *top* of elements in this case, it is conceptually easier to think of head flags as situated *between* two elements, the first element of the current segment and the last element of the previous segment (going from left to right). This distinction becomes important in the case of backward segmented scan where simply flipping the flags is conceptually different from doing the segmented scan from right to left. In this case, the flags must be shifted right by one.

```
flag: 0 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8

flip-flag: 0 0 0 0 1 0 0 0
flip-data: 8 7 6 5 4 3 2 1
```

We see that flipping the flag and the data isn't the same as walking from right to left since the segment should start from 3 but instead starts from 4. Hence the flags need to be shifted one position to the right after flipping. Now we can use the segmented scan algorithm to do backward segmented scan by flipping the final result at the end.

Efficient storage of flags Since flags are Boolean values, it is space-inefficient to store them as 4-byte words. We originally intended to pack 32 flags into a single integer. However, the read-modify-write semantics of parallel threads in CUDA preclude this approach (Section 5).

Instead, we store one flag per byte as shown in the figure above. To reduce shared memory bank conflicts, we allocate flags in chunks of 32 contiguous 4-byte words with four flags per word. Our choice of 32 reflects the 32-thread SIMD warp size in NVIDIA 8-Series GPUs. We stripe the flags across the 32 words in a chunk with a spacing of four bytes between consecutive flags, wrapping to the beginning of the chunk every 32 flags.

Our implementation may benefit from other head flag representations. Blelloch proposes two [Ble90]: a vector of segment lengths and a vector of head pointers (similar to `rowPtr` in Section 4.2). Head flags have two advantages over these representations; first, they are associated directly with each element rather than being stored in a separate data structure; second, these separate data structures have a different size from the vector of data elements and are thus harder to parallelize across thread blocks. Nonetheless, the difficulties we had with storing and computing head flags motivate continued investigation into alternate segmented representations.

Multi-block segmented scan In contrast to unsegmented scan, segmented scan results cannot be propagated across blocks by a uniform add. For segmented scan, the add operation operates on only the first segment of each block. To make this work, we implement segmented scan using separate reduce and down-sweep CUDA kernels.

At the end of the reduce step, we write the partial sum tree and the partial OR tree to global memory. This saves the state so that the down-sweep step can be started later. At this point we do a second-level segmented scan. The inputs to the second-level segmented scan are the flags and data from the last element of the partial sum and partial OR tree of each block. In addition, the second-level segmented scan takes as a third input the first element of the flag vector of each block[‡]. Thus if we have *B* blocks, each input vector is *B* elements long. This multi-block data movement is shown in Figure 1 by the arrows leading out of i and j, which are the last elements of their respective blocks.

The down-sweep phase of the top-level segmented scan is run upon completion of the second-level segmented scan. The state that was saved at the end of the reduce phase is reloaded from global memory. Instead of assigning 0 to the last element of each block (as in unsegmented scan), it is assigned the corresponding element from the output of the second-level segmented scan. For example, the last element of the third block is set to the third element of the output of the second-level segmented scan. This data movement is shown by the arrows leading out of the second-level segmented scan stage into x and k, which are the last elements of their respective blocks. We now proceed with the down-sweep as in the single-block case. The multi-block segmented scan algorithm is summarized in Algorithm 5.

> 1: Perform reduce on all blocks in parallel
> 2: Save partial sum and partial OR trees to global memory
> 3: Do second-level segmented scans with final sums
> 4: Load partial sum and partial OR trees from global memory to shared memory
> 5: Set last element of each block to corresponding element in the output of second-level segmented scan
> 6: Perform down-sweep on all blocks in parallel

Algorithm 5: A multi-block segmented scan algorithm.

2.3. Primitives Built Atop Scan

The scan primitives are used to implement several higher-level primitives that are well-suited as building blocks for general-purpose tasks [Ble90]. Below, we outline the primitives that we have implemented for use in the applications we describe in Section 4.

2.3.1. Enumerate

Enumerate's inputs are a vector and a true/false value per element in the vector. Enumerate can be used with either segmented or non-segmented inputs. The output for each input element is a count of the number of true elements to the left of that element:

```
enumerate([t f f t f t t]) = [0 1 1 1 2 2 3]
```

We implement enumerate by setting a 1 for each true element and a 0 for each false element in a temporary vector, then (exclusive) scanning that vector.

Enumerate is useful in pack (compact) operations, where we only wish to keep elements marked as true. For each true element, the output of enumerate is the address to which the output must be scattered [Hor05].

2.3.2. Distribute (copy)

Like enumerate, distribute can be used with either segmented or non-segmented inputs; it can also be performed in either a forward or backward direction. Distribute copies the element at the head (or tail) of the segment to all other elements in that segment:

```
distribute([a b c][d e]) = [a a a][d d]
```

For segmented inputs, we implement distribute by setting all non-head elements to 0 in a temporary vector and performing a segmented inclusive scan on that vector. For non-segmented inputs, it is more efficient for one thread to write the head element into shared memory and all other threads to read that shared element.

2.3.3. Split and split-and-segment

Like enumerate, split takes two inputs: a vector of elements and a vector of true/false values. Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right. Split-and-segment operates on segmented inputs and performs a stable independent split on each segment, additionally dividing each segment into two segments, one for the falses and one for the trues:

```
split-and-segment([at bf ct][df et ff]) =
              [bf][at ct][df ff][et]
```

Blelloch implements split with two enumerates (requiring two scans), one for the falses going forward, and another for the trues going backward. (Note the first half of the algorithm is essentially just like pack.) To minimize the number of scans, we instead reduce that to one enumerate and perform additional computation to derive the true addresses as follows[§].

```
[t f t f f t f]  # in
[0 1 0 1 1 0 1]  # e = set 1 in false elts.
[0 0 1 1 2 3 3]  # f = enumerate with false=1
            4    # add two last elts. in e, f
                 # == total # of falses
                 # set as shared variable NF
[0 1 2 3 4 5 6]  # each thread knows its id
[4 5 5 6 6 6 7]  # t = id - f + NF
[4 0 5 1 2 6 3]  # addr = e ? f : t
[f f f f t t t]  # out[addr] = in (scatter)
```

```
[5 3 7 4 6]   # initial input
[5 5 5 5 5]   # distribute pivot across segment
[f f t f t]   # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
```

Figure 2: *This quicksort example requires two passes, with three segmented scans per pass (one for the pivot distribute, two within split-and-segment).*

3. Experimental Methodology

For our results, we used an NVIDIA GeForce 8800 GTX GPU connected via PCI Express 16x to an Intel Xeon 3.0 GHz CPU. Our applications ran atop Windows XP with NVIDIA driver version 97.73 and the beta release of the CUDA Toolkit (version dated 14 February 2007).

Unless otherwise noted, timing information is specified for GPU computation only and does not include transfer time to or from the GPU. We feel this best reflects the use of the primitives we describe here, which we expect will be used as components in larger GPU-based computations and would rarely stand alone. To amortize startup costs, we typically run each computation serially many times and present an average runtime for each computation.

3.1. Impact of G80 Hardware

NVIDIA 8-Series GPUs deliver new hardware functionality that adds new capabilities to the GPGPU toolbox. From the perspective of this work, the two most important are the addition of 16 kB of shared memory per multiprocessor and the support of generalized scatter within the GPU programmable units.

Shared memory The addition of shared memory enables more efficient data sharing between threads and improves overall performance by eliminating memory traffic to main GPU memory. However, shared memory does not introduce any new functionality that was not available on previous hardware.

Scatter Scatter, on the other hand, does introduce functionality not present on older NVIDIA GPUs. While the scan and segmented scan primitives could be implemented on older hardware, scatter enables higher performance and storage efficiency. Scatter is efficient but not necessary for such operations as pack; when the vector of scanned data items is ordered, as in the `enumerate` operation used by pack, Horn's gather-search operation [Hor05] emulates scatter at the cost of $\log n$ passes. Buck summarizes other methods for implementing scatter on older GPUs [Buc05].

Scatter functionality is available in the R520 and R600 families of AMD GPUs [PSG06]. Thus we expect that the techniques we describe are also directly applicable to AMD GPUs.

4. Applications

4.1. Quicksort

Quicksort is a popular, efficient primitive for sorting on serial machines. Its control complexity and irregular parallelism have prevented previous implementations on GPUs, but the segmented scan primitive leads to an elegant formulation credited to Blelloch [Ble90].

Our algorithm runs in parallel over all segments in the input. All communication between elements (threads) in the algorithm is within a single segment, so the segmented scan primitives are an ideal fit. We begin by choosing a pivot element in each segment (we choose the first element in the segment) and then distributing that pivot across the segment. We then compare the input element to the pivot. On alternating passes through the algorithm we compare either greater-than or greater-than-or-equal. The comparison produces a segmented vector of trues and falses, which we use to split-and-segment the input; smaller elements move to the head of the vector and larger elements to the end. Each segment splits into two¶. We begin with a single segment that spans the entire input and finish when the output is sorted, which we check with a global reduction after each step. Figure 2 shows a small example.

4.2. Sparse Matrix-Vector Multiply

Matrix-based numerical computations are a good match for the GPU for two reasons: first, they are computationally intense, and second, they exhibit substantial parallelism. Representative GPGPU efforts focusing on matrix computations include Larsen and McAllister's 2001 dense-matrix multiplication study [LM01], Moravánszky's dense matrix representation [Mor02], or Krüger and Westermann's general linear algebra framework [KW03].

Sparse matrices are key components of many important numerical computing algorithms, including singular value

---

¶ Instead of alternating comparisons and a 2-way split, Blelloch always uses the same comparison and uses a 3-way split instead. We found the additional control complexity of a 3-way split, and the additional number of segmented scans it required, did not justify its fewer number of passes overall.

decomposition, conjugate gradient, and multigrid. The appeal of a sparse matrix representation is less storage and computation than its dense cousin. The ideal sparse matrix representation only stores the non-zero elements and requires neither padding (which requires extra storage space) nor sorting (which requires additional computation). However, the irregularity of sparse matrices makes parallelizing operations on them difficult.

The most notable current work in this area on the GPU is from Bolz et al. [BFGS03]. Their "jagged diagonal" formulation for sparse matrices [AS89] is simpler to parallelize than a truly sparse representation, but requires a sort of its rows to place them in descending order by length. Krüger and Westermann render a separate point for every four non-zero entries in a row [KW03]. Brook's `spMatrixVec` test uses a compressed sparse row format but runs in parallel on rows, so the runtime on each row is proportional to the largest number of elements in any row [BH03].

We choose the compressed sparse row (CSR) format for our sparse matrices. CSR requires neither preprocessing nor padding and is one of the representations of choice in the numerical computing community. Our CSR representation and sparse-matrix-vector multiplication algorithm both roughly follow that of Blelloch et al. [BHZ93].

We represent an $n \times n$ CSR sparse matrix containing e non-zero elements (entries) with the following three data structures:

1. The e-element `value` vector contains all e non-zero elements (entries) in the matrix, read in scan order (left to right, top to bottom). We store this as a float vector.
2. The e-element `index` vector contains an integer identifying the column for each element in the `value` vector.
3. The n-element `rowPtr` vector contains an integer index into `value` that points to the first element in each row.

The matrix consists of these three data structures. We multiply it by a vector `x` of length n and add the result to a vector `y` of length n. With these data structures in main GPU memory, and additional `flag` and `product` temporary data structures with e entries each, matrix multiplication then proceeds in four steps. We show an example in Figure 3.

1. The first kernel runs over all entries. For each entry, it sets the corresponding `flag` to 0 and performs a multiplication on each entry: `product = x[index] * value`.
2. The next kernel runs over all rows and sets the head flag to 1 for each `rowPtr` in `flag` through a scatter. This creates one segment per row.
3. We then perform a backward segmented inclusive sum scan on the e elements in `product` with head flags in `flag`.
4. To finish, we run our final kernel over all rows, adding the value in `y` to the gathered value from `products[idx]`.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} += \begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\text{value} = [a,b,c,d,e,f]$$
$$\text{index} = [0,2,0,1,2,2]$$
$$\text{rowPtr} = [0,2,5]$$

$$\text{product} = [x_0 a, x_2 b, x_0 c, x_1 d, x_2 e, x_2 f] \quad (1)$$
$$= [[x_0 a, x_2 b][x_0 c, x_1 d, x_2 e][x_2 f]] \quad (2)$$
$$= [[x_0 a + x_2 b, x_2 b]$$
$$[x_0 c + x_1 d + x_2 e, x_1 d + x_2 e, x_2 e][x_2 f]] \quad (3)$$
$$y = y + [[x_0 a + x_2 b, x_0 c + x_1 d + x_2 e, x_2 f] \quad (4)$$

Figure 3: *Segmented scan can be used to perform sparse-matrix vector multiplication. Our goal is to solve the top equation; we represent the sparse matrix with the three vectors* value, index, *and* rowPtr. *The four computation steps at bottom, described in Section 4.2, compute the sparse-matrix-vector product. Our formulation is more efficient than previous methods, which required either sorting* value *by rows [BFGS03] or wasting work when rows were not of uniform length [BH03].*

4.3. Tridiagonal Matrix Solvers and Fluid Simulation

The reduce/downsweep structure of the scan primitive can also be applied to other problems, including the solution of tridiagonal systems $y = Tx$, where the tridiagonal matrix T has entries only along the main diagonal and the adjacent diagonals. Kass and Miller demonstrated the use of tridiagonal systems for fluid simulation using a shallow-water assumption [KM90], and Kass et al. later used a similar computation for real-time depth-of-field [KLO06]. The latter work was the first to use the technique of cyclic reduction to solve tridiagonal systems on the GPU. Here we implement a GPU-based fluid simulation algorithm using the method of Kass and Miller with a GPU scan-based tridiagonal solver. This inexpensive, physically accurate method for shallow fluid simulation is a viable alternative to other PDE solution methods [KW03]. Our implementation leverages our scan framework for the related problem of cyclic reduction and demonstrates that our CUDA-based code can be easily integrated into a graphics application.

Following the treatment of Kass and Miller, in our water simulation (Plate 1), we describe the water surface as an $n \times m$ 2D array of heights. We use the alternating direction method to first solve a tridiagonal system for each of n rows in parallel, then for each of m columns in parallel. Figure 4 shows the difference in the reduce step communication pattern between scan and the tridiagonal solver. Because the tridiagonal solver requires more communication, it is more
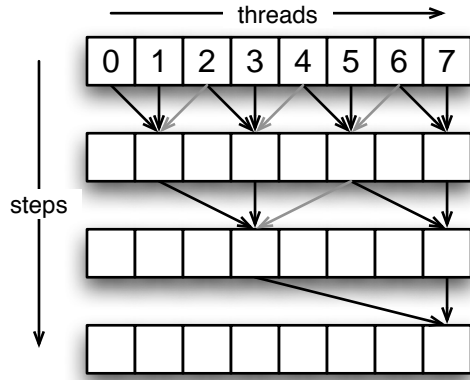
Figure 4: *The communication pattern between threads in the reduce step of a tridiagonal solver is similar to scan. Black arrows show communication used by both scan and tridiagonal solve; gray arrows show additional communication needed for tridiagonal solve. The down-sweep communication requirements are also more complex for the tridiagonal solver.*

Figure 5: *Performance results for four types of scan. The most meaningful comparison between the types is between scan and the two segmented scans; the "fast" scan processes eight elements per thread rather than two and is provided for reference [HSO07].*

difficult to divide across blocks. Our implementation can process grids of up to 512×512 elements with each line of the grid processed in shared memory by a single thread block. For larger simulations, we divide each line of the input across multiple thread blocks and use global memory rather than shared memory to communicate between threads.

5. Results and Analysis

Scan vs. Segmented Scan For 1M elements, with 128 threads (256 elements) per thread block and using the same formulation for all scans, a forward unsegmented scan takes 0.79 ms, a backward unsegmented scan takes 0.88 ms, a forward segmented scan takes 2.61 ms, and a backward segmented scan takes 4.29 ms. The runtime cost is linear starting at roughly 4k elements, which corresponds to 16 active thread blocks (the GeForce 8800 GTX has 16 multiprocessors). Figure 5 shows performance results for four scan variants up to 8M elements.

One of the most difficult aspects of our segment implementation is the representation of segment head flags. Allocating more than a bit per flag is storage-inefficient, but the lack of atomic read-modify-write shared memory instructions in CUDA preclude optimal packing. It could be implemented at additional cost with a per-warp parallel reduction, which we plan to try in the future. Hardware support for a packed flag representation in general would also solve this problem. Another possible hardware solution, albeit one that is fairly special-purpose, is associating an extra hardware bit with each register (essentially a 33-bit wide data path), which may have applications beyond segmented scan. Chatterjee et al.'s results on the Cray Y-MP show that effi-
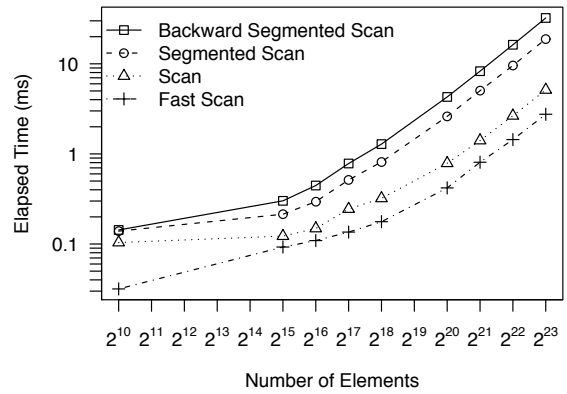
cient hardware support for flags makes their segmented scan implementations "only marginally more expensive than the unsegmented versions" [CBZ90].

Large segmented scans are about three times slower than large unsegmented scans for two reasons. First, segmented scan does more I/O from global memory than scan. The flag vector and the partial OR tree are read twice from global memory, once at the beginning of the reduce phase and again at the beginning of the down-sweep phase. Both are also written when the state is saved at the end of the reduce phase. Segmented scan also does more computation than scan. In the reduce phase the flags are logically ORed, and in the down-sweep phase segmented scan has an if-then-elseif construct. The segmented scan implementation does significantly more address calculation to pack and unpack flags from the striped compact representation.

Backward segmented scans are about two times slower than forward segmented scans. This is because of uncoalesced reads and writes of the flag vector and the partial OR tree from global memory. To implement backward scan, we read the data in reverse order from shared memory. However, this violates coalesced I/O restrictions imposed by the GPU since threads access memory in decreasing order. In the future we will coalesce the I/O of flags by reading them in increasing order from global memory and reversing them in shared memory. This will result in similar performance for backward and forward segmented scan, which will in turn improve the performance of sparse matrix-vector multiply.

Sparse Matrix-Vector Multiply Our sparse matrix-vector multiply performance approaches, but does not yet meet, the throughput of a best-of-breed CPU implementation. For evaluation we tested the medium-sized "raefsky2" ma-

trix [Dav94], a 3242×3242, 294,276-element sparse matrix that represents incompressible flow in a pressure-driven pipe. Using standard accounting for this operation, we achieve 215 MFLOPS on matrix-vector multiplication (365 matrix-vector multiplies per second), compared to the highly-optimized, self-tuning "oski" CPU implementation's December 2006 results of 522 MFLOPS on a Pentium 4 and 294 MFLOPS on an AMD Opteron [Gah06]. Most of our runtime is spent in the backward segmented scan operation and we are hopeful that continued improvements in our implementation will yield further performance gains.

Comparison to previous results is difficult because they were implemented on older hardware, but as a comparison, Bolz et al.'s 2003 implementation on a 500 MHz NVIDIA GeForce FX (15 GFLOPS, 12.8 GB/s bandwidth) achieved 120 matrix-vector multiplies per second (not including the sort time) on a 37k-entry sparse matrix [BFGS03], a factor of 24 slower than our implementation on newer (345 GFLOPS, 86 GB/s) hardware.

Sort We compared 5 sort implementations: split-based radix sort per block, followed by a parallel merge sort of blocks [HSO07]; quicksort per block, followed by the parallel merge; a (global) split-based radix sort across all inputs (no merge necessary); and 2 CPU-based sorts using STL's sort and C's quicksort routines. For a sort of 4M 32-bit integers, runtimes follow.

Test	Runtime (ms)
GPU global radix	165.0 ms
GPU radix/merge	317.8 ms
CPU STL sort	571.7 ms
CPU quicksort	908.8 ms
GPU quicksort/merge	2050.3 ms

These results parallel existing literature; Govindaraju et al. concluded that GPU sorts were appreciably but not massively faster than CPU-based sorts [GGKM06], and Blelloch finds "the most practical parallel sorting algorithm" (bitonic sort) and split-based radix sort had similar performance on the Connection Machine [Ble90]. Our interest in this paper is more oriented to demonstrating a new GPU-based sorting algorithm, namely quicksort. Quicksort has an excellent expected complexity of $O(n \log n)$. On scalar CPUs, quicksort is in theory and in practice superior to bitonic sort's $O(n \log^2 n)$ and in practice nearly always better than radix sort's $O(b)$ passes. On the GPU, however, quicksort is a poor performer for several reasons.

High-performance sorts are typically bound by bandwidth, not compute, but our quicksort implementation is very definitely bound by compute. In particular, the bookkeeping instructions to manage multiple active regions in shared memory and the staging of regions to and from the bank-conflict-free representations used by the segmented scan together result in both lengthy programs and a large number of active registers. Long programs are slow, and using many registers reduces the occupancy of the program in the multiprocessors (according to the profiler, the occupancy is only 1/6). In the long term, as compute costs become cheaper compared to communication costs, complex programs with large register usage may become more attractive.

We would like to extend our sort implementations to support key-value sorts for better comparison with previous GPU sorts. The bitonic sort in the NVIDIA CUDA SDK is single-block only and was thus not suitable for comparison.

Tridiagonal Solver Our tridiagonal solver has little difficulty maintaining real-time performance for shallow water simulation. For the 128×128 simulation pictured in the middle row of Plate 1, we measure 1207 simulation steps per second for compute only. The compute time is considerably less than the overhead of mapping and unmapping the vertex buffer(roughly 4:1 overhead:compute). We also compared our 512×512 solver against a CPU cyclic reduction solver (which is not the serially optimal CPU solver). Running from GPU main memory, the CUDA implementation was 3 times faster than the CPU, and using shared memory, 12 times faster (2.3 ms vs. 27.6 ms).

6. Conclusion

One of the difficulties with GPGPU programming has been the vertical nature of GPGPU program development. With few exceptions, most applications are developed by a single team from the API interface to the hardware up to the application itself, with little to no code reuse from other projects. The field would benefit from a more horizontal model of program development with libraries of GPGPU primitives available for use by GPGPU applications allowing code reuse and factorization by GPGPU developers.

What should those primitives be? This is a broad and important question facing the GPGPU community. For a highly parallel machine such as the GPU, we believe it is important not just to consider traditional scalar primitives but also primitives that were designed for parallel programming environments and machines. The scan primitives are not particularly well suited for scalar machines, but they are an excellent match for a broad set of problems on parallel hardware generally and, we believe, specifically the GPU. We expect that continued investigation of a wide range of potential parallel primitives, and optimized implementations of these primitives, will provide insight for future hardware and software for GPGPU and provide more productive GPGPU programming environments for future developers.

Lengyel generated the images for the shallow water simulation using his C4 game engine.

References

[AS89] ANDERSON E., SAAD Y.: Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing 1*, 1 (May 1989), 73–95.

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22*, 3 (July 2003), 917–924.

[BH03] BUCK I., HANRAHAN P.: *Data Parallel Computation on Graphics Hardware*. Tech. Rep. 2003-03, Stanford University Computer Science Department, Dec. 2003.

[BHZ93] BLELLOCH G. E., HEROUX M. A., 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.

[Ble90] BLELLOCH G.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Buc05] BUCK I.: Taking the plunge into GPU computing. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 32, pp. 509–519.

[CBZ90] CHATTERJEE S., BLELLOCH G. E., ZAGHA M.: Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 Conference on Supercomputing* (1990), pp. 666–675.

[Dav94] DAVIS T. A.: The University of Florida sparse matrix collection. *NA Digest 92*, 42 (16 Oct. 1994). http://www.cise.ufl.edu/research/sparse/matrices.

[Gah06] GAHVARI H. B.: *Benchmarking Sparse Matrix-Vector Multiply*. Master's thesis, University of California, Berkeley, Dec. 2006.

[GGK06] GRESS A., GUTHE M., KLEIN R.: GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum 25*, 3 (Sept. 2006), 497–506.

[GGKM06] GOVINDARAJU N. K., GRAY J., KUMAR R., MANOCHA D.: GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (June 2006), pp. 325–336.

[Hor05] HORN D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 36, pp. 573–589.

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast summed-area table generation and its applications. *Computer Graphics Forum 24*, 3 (Sept. 2005), 547–555.

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, Aug. 2007, ch. 31.

[Ive62] IVERSON K. E.: *A Programming Language*. Wiley, New York, 1962.

[KLO06] KASS M., LEFOHN A., OWENS J.: *Interactive Depth of Field Using Simulated Diffusion on a GPU*. Tech. Rep. #06-01, Pixar Animation Studios, Jan. 2006. http://graphics.pixar.com/DepthOfField/.

[KM90] KASS M., MILLER G.: Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Aug. 1990), pp. 49–57.

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3 (July 2003), 908–916.

[LKS*06] LEFOHN A. E., KNISS J., STRZODKA R., SENGUPTA S., OWENS J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics 26*, 1 (Jan. 2006), 60–99.

[LM01] LARSEN E. S., MCALLISTER D.: Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Nov. 2001), p. 55.

[Mor02] MORAVÁNSZKY A.: Dense matrix algebra on the GPU. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, Engel W. F., (Ed.). Wordware Publishing, 2002, pp. 352–380.

[NVI07] NVIDIA CORPORATION: NVIDIA CUDA compute unified device architecture programming guide. http://developer.nvidia.com/cuda, Jan. 2007.

[PSG06] PEERCY M., SEGAL M., GERSTMANN D.: A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications* (Aug. 2006).

[Sch80] SCHWARTZ J. T.: Ultracomputers. *ACM Transactions on Programming Languages and Systems 2*, 4 (Oct. 1980), 484–521.

[SLO06] SENGUPTA S., LEFOHN A. E., OWENS J. D.: A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures* (May 2006), pp. D–26–27.
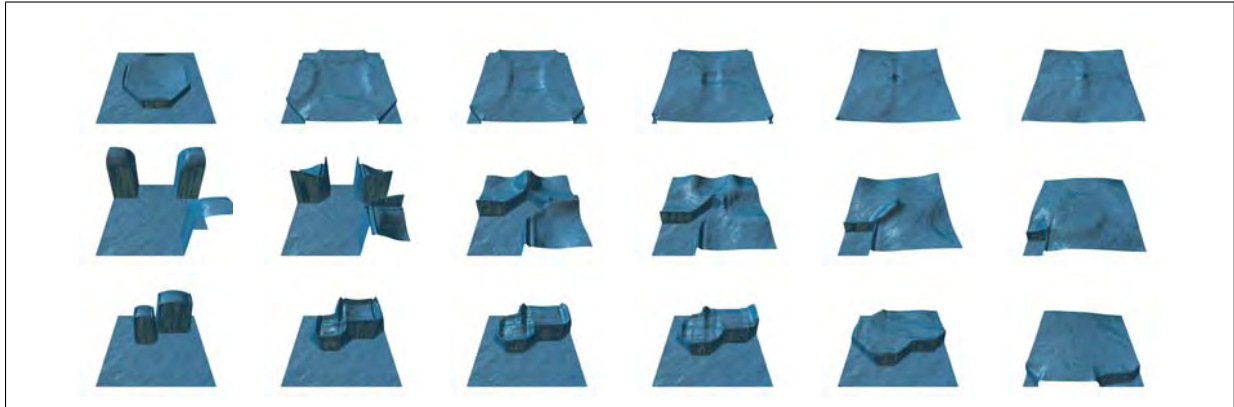
Plate 1: The pictures above are frames from rendering our shallow-water simulation, which uses our GPU-based tridiagonal matrix solver. Each row displays several frames from a different simulation, with the starting point of the simulation at the left. The top row simulates a 256×256 grid, the middle row a 128×128 grid, and the bottom row a 64×64 grid. Thanks to Eric Lengyel for generating these images using his C4 game engine.