Book title goes here

List of Figures

List of Tables

Chapter 1

Efficient Parallel Scan Algorithms for Many-core GPUs

Shubhabrata Sengupta

University of California, Davis

Mark Harris

NVIDIA Corporation

Michael Garland

NVIDIA Corporation

John D. Owens

University of California, Davis

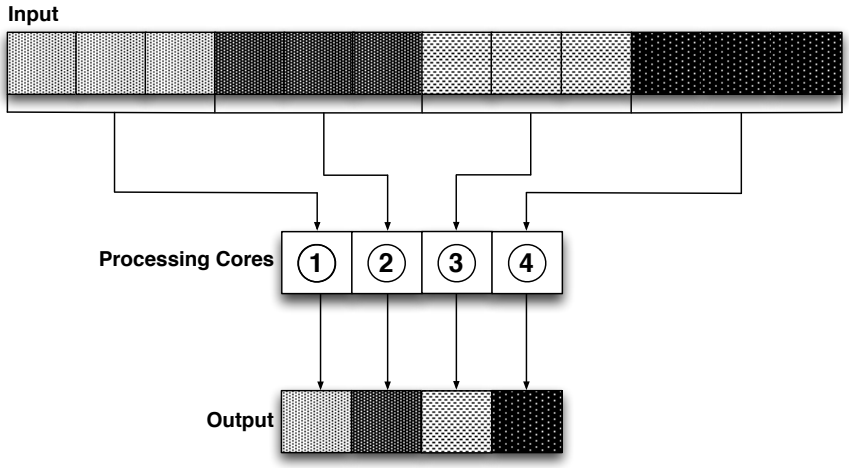

FIGURE 1.1: A simple memory access pattern in which each processor reads a contiguous bounded neighborhood of input (each neighborhood has a different hatching pattern) and produces one output item.

1.1 Introduction

We have witnessed a phenomenal increase in computational resources for graphics processors (GPU) over the last few years. The highest performing graphics processors from both ATI and NVIDIA already have billions of transistors, resulting in more than a teraflop of peak processing power. This incredible processing power comes from the presence of hundreds of processing cores, all on the same chip.

This computational resource has historically been designed to be best suited for the *stream programming model*, which has many independent threads of execution. In this model a single program operates in parallel on each input, producing one output element for each input element. This model extends nicely to problems in which each output element depends on a small, bounded neighborhood of inputs. This simple streaming memory access pattern is shown in Figure 1.1.

Many interesting problems (sorting, sparse matrix operations) require more general access patterns in which each output may depend on a variable number of inputs. In addition, the ratio between the number of input elements and the number of output elements may not be constant. This implies that processing cores may produce a variable number (including zero) of elements.

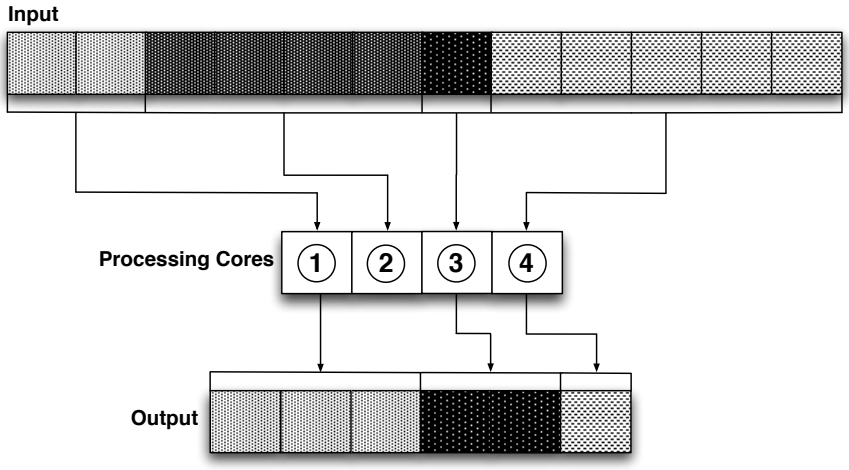

FIGURE 1.2: A more general memory access pattern in which each processor reads input of variable length and produces output of variable length. Processor 2 produces no output.

Two different scenarios may occur: *expansion*, in which a smaller number of input elements produces a large number of output elements; and *contraction*, in which a larger number of input elements produces a smaller number of output elements. Figure 1.2 shows a variable input/output memory access pattern. Elements being processed by processor core 1 undergo an expansion while those processed by cores 2, 3 and 4 undergo a contraction.

This is a challenging scenario for fine-grained parallelism, since we would like each processor core to work independently on its section of the input, and also to write its output independently into the output array. To allow this parallel execution, the key question is "where in the output array does each processor core write its data?". To answer this question, each processor core needs to know where the processor core to its left will finish writing its data, resulting in an apparent serial dependency.

A recurrence relation is yet another common case in which a serial dependency arises. This is expressed in the following loop.

```
for all i
    B[i] = f(B[i-1], A[i])
```

In this case each result element `B[i]` depends on all elements to its left.

In Section 1.3 we describe a family of algorithmic primitives, the *scan* primitives, that let us solve such seemingly serial problems efficiently in the data-parallel world. But first it is helpful to look at our programming environment, so that we can present code snippets along the way.

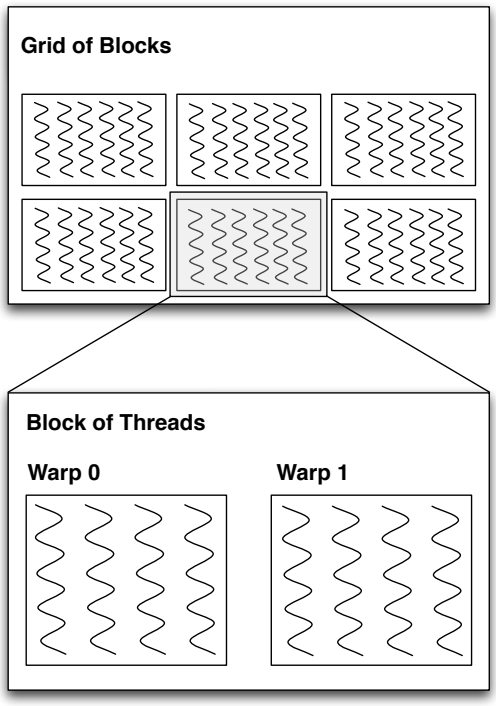

FIGURE 1.3: In the hierarchical architecture of the CUDA programming model, a grid is composed of multiple thread blocks, each of which is composed of multiple warps of threads (squiggly lines). This diagram shows only four threads per warp for simplicity.

1.2 CUDA—a general-purpose parallel computing architecture for graphics processors

We chose NVIDIA's CUDA GPU Computing environment [14, 15] for our implementation. CUDA provides a direct, general-purpose C language interface to the programmable processing cores (also called Streaming Multiprocessors or SMs) on NVIDIA GPUs, eliminating much of the complexity of writing non-graphics applications using graphics APIs such as OpenGL.

CUDA executes GPU programs using a grid of *thread blocks* of up to 512 threads each. Each thread executes functions that are either declared `__global__`, which means they can be called from the CPU, or `__device__`, which means they can be called from other `__global__` or `__device__` func-

tions. The host program specifies the number of thread blocks and threads per block, and the hardware and drivers map thread blocks to processing cores on the GPU. Within a thread block, threads can communicate through shared memory and cooperate using simple thread synchronization. Threads are scheduled on processor cores in groups of 32 called warps. All threads in a warp execute one instruction at a time in lockstep. This hierarchy is shown in Figure 1.3. Another popular programming environment for parallel processors, OpenCL [13], has an identical hierarchy, albeit with different terminology.

1.3 Scan: an algorithmic primitive for efficient data-parallel computation

Scan is an algorithmic primitive that solves a wide class of interesting problems on processors that have been designed to provide maximum performance for streaming access [2, 3]. These operations are the analogs of parallel prefix circuits [11], which have a long history, and have been widely used in collection-oriented languages dating back to APL [10].

Even though the scan primitive is conceptually quite simple, it forms the basis for a surprisingly rich class of algorithms. These include various sorting algorithms (radix, quick, merge), computational geometry algorithms (closest pair, quickhull, line of sight), graph algorithms (minimum spanning tree, maximum flow, maximal independent set) and numerical algorithms (sparse matrix-dense vector multiply) [2]. They also form the basis for efficiently mapping nested data-parallel languages such as NESL [4] on to flat data-parallel machines. In the following sections we look at scan and its segmented variant.

1.3.1 Scan

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 $b_i = I \oplus \cdots \oplus a_{i-1}$. Some common binary associative operators are add (prefix-sum), min (min-scan), max (max-scan) and multiply (mul-scan).

As a concrete example, consider the input sequence:

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

[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>
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 1.4: Serial implementation of inclusive scan for generic operator `OP` over values of type `T`.

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 for the operator, which in this case is 0.

1.3.1.1 A serial implementation

Implementing scan primitives on a serial processor is trivial, as shown in Figure 1.4. Note that throughout this paper, we use C++ templates to make scan generic over the operator `OP` and the datatype of values `T`.

1.3.1.2 A basic parallel implementation

Figure 1.5 shows a simple CUDA C implementation of a well-known parallel scan algorithm [5, 9]. This code scans the binary operator `OP` across an array of $n = 2^k$ values using a single thread block of 2^k threads. We make two assumptions—the number of values is a power of 2 and each thread inputs exactly 1 value—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 (`__syncthreads()`).

1.3.2 Segmented Scan

Segmented scan generalizes the scan primitive by simultaneously performing separate parallel scans on *arbitrary* contiguous partitions ("segments") of

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

    // number of threads in block
    unsigned int n = blockDim.x;

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

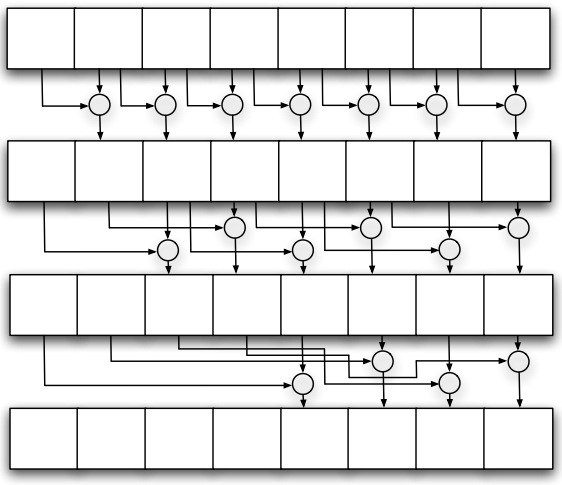

FIGURE 1.6: We show the memory access pattern for the code snippet shown in Fig 1.5. Arrows show the movement of data. Circles show the binary operation. Since our input has 8 elements we need 3 iterations of the `for` loop.

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 that stores a 1 for each element that begins a segment and 0 for all others. This representation is convenient for massively parallel machines. All other representations (for example, every element knows the index of the first and/or the last index of its own segment) 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 [2, 16]. Given the operator $\oplus$ we can construct a new operator $\oplus^s$ that operates on flag-value pairs (f_x, x) as follows:

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

Segmented scan can also be implemented directly rather than by operator transformation [6, 17]. In Section 1.5 we explore both of these implementation strategies.

In the next two sections, we turn to the question of how to efficiently implement the scan primitives on modern graphics processors.

1.4 Design of an efficient scan algorithm

1.4.1 Hierarchy of the scan algorithm

The most efficient CUDA code respects the block granularity imposed by the CUDA programming model and the warp granularity of the underlying hardware. For example, threads within a block can efficiently cooperate and share data using on-chip shared memory and barrier synchronization. Threads within a warp, however, do not need explicit barrier synchronization in order to share data because they execute in lockstep.

We organize our algorithms to match the natural execution granularities of warps and blocks in order to maximize efficiency. 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 parallel 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.

To simplify the discussion, we assume that there is exactly 1 thread per element in the sequence being scanned. 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.

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_warp(volatile T *ptr,
                       const uint idx=threadIdx.x)
{
    // index of thread in warp (0..31)
    const uint lane = idx & 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 1.7: Scan routine for a warp of 32 threads with operator `OP` over values of type `T`. The `Kind` parameter is either `inclusive` or `exclusive`.

1.4.2 Intra-Warp Scan Algorithm

We begin by defining a routine to perform a scan over a warp of 32 threads, shown in Figure 1.7. It uses precisely the same algorithm as shown in Figure 1.5, 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 [2]. However, since the threads of a warp execute in lockstep, there is actually no advantage in decreasing work at the expense of increasing 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 [2] performs twice as many steps as the algorithm used here, it leads to measurably lower performance in practice.

The removal of explicit barrier synchronization from warp-synchronous

code has an unintended consequence. Without synchronization barriers, an optimizing compiler may choose to keep data in registers rather than writing it to shared memory, causing cooperating threads in a warp to read incorrect values from shared memory. Therefore shared variables used without synchronization by multiple threads in a warp must be declared **`volatile`** to force the compiler to store the data in shared memory, as shown in Figures 1.7, 1.10, and 1.11.

1.4.3 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. With a single warp, perform `scan_warp()` on the partial results from Step 2.
4. For each thread of warp i, accumulate the partial results from Step 3 into that thread's 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 1.8. The individual steps are labeled and correspond to the algorithm outline.

1.4.4 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.

1. Scan all blocks in parallel using `scan_block()`.
2. Store the partial result (last element) from each block i to `block_results[`i`]`.
3. Perform a scan of `block_results`.
4. Each thread of block i combines element i from Step 3 to its output element from Step 1. The combination simply uses the binary operator for the scan operation. For example, if the operation is add, the element i from Step 3 is added to all output elements from Step 1.

```
template<class OP, ScanKind Kind, class T>
__device__ T scan_block(T *ptr,
                        const uint idx=threadIdx.x)
{
    const uint lane   = idx & 31;
    const uint 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 1.8: Intra-block scan routine composed from `scan_warp()` primitives.

Because they require global synchronization, Steps 1 and 2 are performed by the same CUDA kernel, while Steps 3 and 4 are performed in their own separate kernels; thus we require three separate CUDA kernel invocations. Indeed, Step 3 may require recursive 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.

1.5 Design of an efficient segmented scan algorithm

To implement efficient segmented scan routines, we follow the same design strategy already outlined in Section 1.4 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.

1.5.1 Operator Transformation

As described in Section 1.3.2, segmented scan can be implemented by transforming the operator $\oplus$ into a segmented operator $\oplus^s$ that operates on flag–value pairs [16]. 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 1.9. This trivially converts the inclusive `scan_warp()` and `scan_block()` routines given in Section 1.4 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 1.10. The structure of this procedure is exactly the same as the one shown in Figure 1.7 except that (1) it does roughly twice as many operations

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

and (2) requires slightly different logic for determining the final inclusive/exclusive result.

1.5.2 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 1.11, 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 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 (as defined in Section 1.3.1). 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 1.7 is essentially just the special case where `mindex=0`. Figure 1.12 illustrates an example of the resulting data movement for a warp of size 8.

1.5.3 Block and Global Segmented Scan Algorithms

As we have done in section 1.4.1, to maximize efficiency, we need to organize our algorithms to match the natural execution granularities of warp and block. At the lowest level, we have shown two ways to design an *intra-warp* primitive to perform a segmented scan across a single warp of threads. We can

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_warp(volatile T *ptr,
                          volatile flag_type *hd,
                          const uint idx = threadIdx.x)
{
   const uint 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 1.10: 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 uint idx = threadIdx.x)
{
    const uint 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 1.11: Intra-warp segmented scan using conditional indexing.

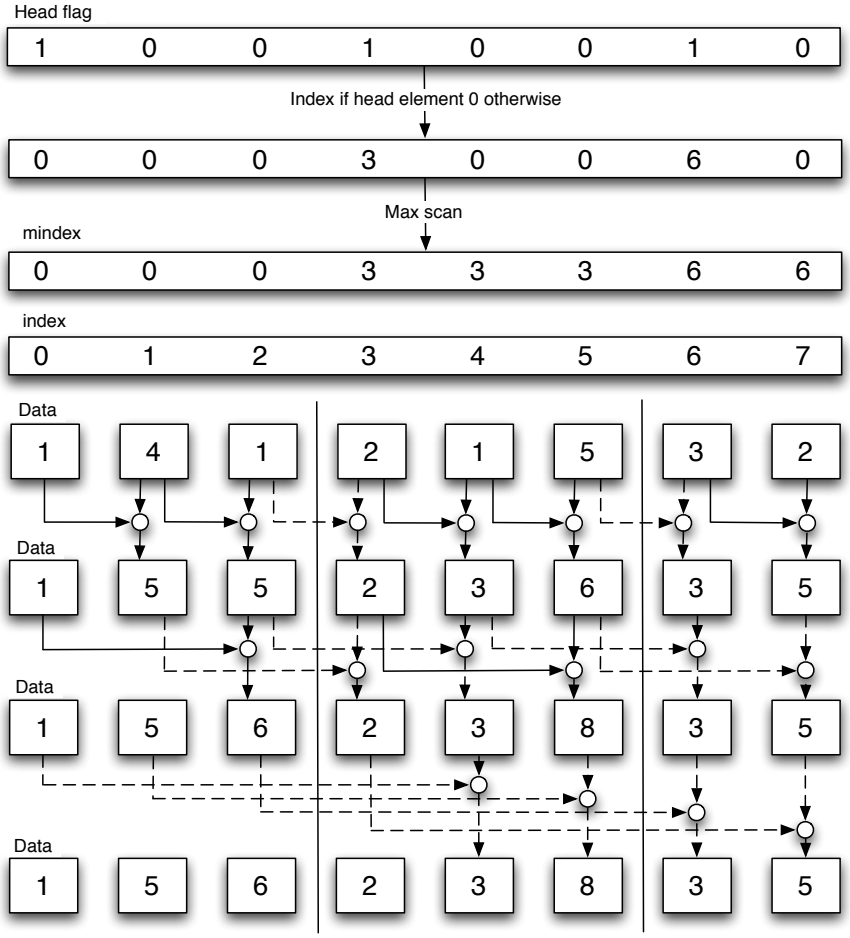

FIGURE 1.12: Data movement in intra-warp segmented scan code shown in Figure 1.11 for threads 0–7 of an 8-thread warp. Data movement in the unsegmented case (dotted arrows) crossing segment boundaries (vertical lines) are not allowed.

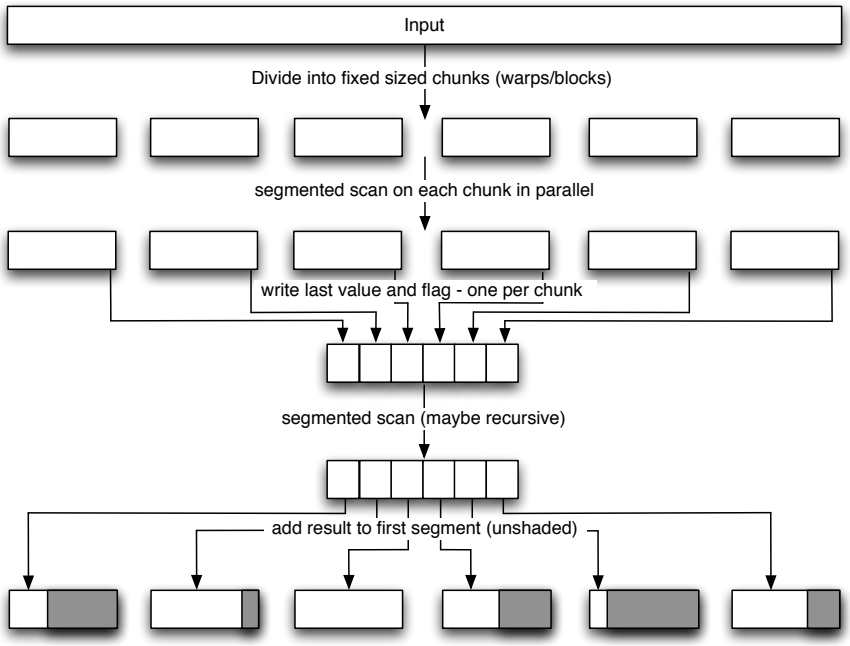

FIGURE 1.13: Constructing block/global segmented scans from warp/block segmented scans. The unshaded part in the last row shows the extent of the first segment.

then use either of these two ways to construct an *intra-block* primitive that composes intra-warp segmented scans together to perform a segmented scan across a block of threads. Finally, we combine grids of intra-block segmented scans into a *global* segmented scan of arbitrary length.

The methodology used for constructing an intra-block segmented scan is essentially identical to our construction of the intra-block scan routine in Section 1.4.3, 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[2] 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 1.13.

The full CUDA implementation of this algorithm is shown in Figure 1.14. We first record if the warp starts with a new segment, because the flags array is

[2]a reduction operation with the logical OR operator. See Section 1.7.1 for an explanation of reduction.

```
template<class OP, ScanKind Kind, class T>
__device__ T segscan_block(T *ptr, flag_type *hd,
                           const uint idx = threadIdx.x)
{
    // Right shift by 5 as warp size is 32
    uint warpid     = idx >> 5;

    uint warp_first = warpid << 5;
    uint 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 1.14: Intra-block segmented scan routine built using intra-warp segmented scans.

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 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 1.14 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.

1.6 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 the complexity of 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.

$$S_b = O(S_w \log_w B) = O\left((\log_2 w)\left(\frac{\log_2 B}{\log_2 w}\right)\right) = O(\log_2 B) \tag{1.1}$$

$$W_b = O\left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil W_w\right) = w \log_2 w \left(\sum_{i=1}^{\log_w B} \left\lceil \frac{B}{w^i} \right\rceil\right) = O\left(B \log_2 w\right) \tag{1.2}$$

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.

$$S_n = O(S_b \log_B n) = O((\log_2 w)(\log_B n)) = O(\log_2 n) \tag{1.3}$$

$$W_n = O\left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil W_b\right) = w \log_2 w \left(\sum_{i=1}^{\log_B n} \left\lceil \frac{n}{B^i} \right\rceil \sum_{j=1}^{\log_w B} \left\lceil \frac{B}{w^j} \right\rceil\right) = O\left(n \log_2 w\right) \tag{1.4}$$

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.

1.7 Some alternate designs for scan algorithms

In this section we show two different ways to structure the data flow among the various levels of a scan implementation. Both work at the intra-block and the inter-block level and they can be combined. They also work for both segmented and unsegmented scans, though for the sake of simplicity we restrict the discussion to the unsegmented version. We also refer readers to the recent technical report by Merrill and Grimshaw [12] with their memory-bandwidth-bound scan implementation and extensive performance analysis.

1.7.1 Saving Bandwidth by performing a Reduction

Dotsenko et al. [7] have showed an alternate strategy for an efficient scan implementation. They too follow a bottom up strategy where the intra-block scan is built from smaller scans (which could be warp-wide or smaller, but rarely larger for efficiency reasons) and the scan of an arbitrary number of elements is based on intra-block scans.

The key contribution of this method is the use of a reduction operation as the first stage. A reduction operation on an array of elements is a binary operator applied to all elements in a "telescopic" fashion. This means the binary operator is applied to the first two elements, the result of which is combined with the third element by the same operator. A simple example is performing a reduce on an array with an add operator that produces the sum of all elements in the array.

To illustrate, we show how a scan of an arbitrary number of elements can be performed using a block wide reduce (`reduce_block()`) and scan (`scan_block()`) primitive.

1. Do a reduction operation on all blocks in parallel using `reduce_block()` and store the result from each block i to `block_results[`i`]`.

2. Perform a scan of `block_results`.

3. One thread of each block i combines element i from Step 3 to its first element from Step 1. The combination just uses the binary operator for the scan operation. For example, if the operation is add, the element i from Step 3 is added to the first output element from block i in Step 1.

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

The same strategy can be used to construct an intra-block scan (`scan_block()`) by doing a reduction and subsequent scan on warp-wide or even smaller set of elements. The reduction and scan at this level is done sequentially: one thread does the reduction and scan operation on all elements of this smallest set.

An interesting thing to note is that this method has to save less state between Step 1 and Step 4. While the method shown in section 1.4.4 has to write out N elements at the end of Step 1, this one only writes out N/B elements where B is the block size. On the other hand this method does more computation since the simple add operation in Step 4 in section 1.4.4 is replaced by a reduction operation in Step 1 here. This is a commonly used optimization strategy on GPUs where off-chip bandwidth is limited while computational resources are rarely so.

On the flip side, a possible disadvantage is that such a combination of reduction then scan may assume that the operator is commutative. This would restrict the operators that can be used for scan to those which are now not only associative but also commutative.

1.7.2 Eliminating recursion by performing more work per block

Step 3 in the method described in section 1.4.4 and Step 2 in the method described in section 1.7.1 require a recursive call to scan. If each block is

performing a scan on B elements, then `block_results` ends up with N/B elements after the first step. If $N/B > TC$, where TC is the number of threads in a block, we have to do a global scan on `block_results` recursively. Otherwise we can use our intra-block scan algorithm (Section 1.4.3) to scan `block_results`, eliminating all further recursive kernel calls and associated data transfer to and from off-chip memory.

Thus given an N we choose B such that $N/B \leq TC$. Given that all our intra-block algorithms assume that each thread processes one element, and given that each block can only have TC threads (TC can take a maximum value of 512 in current architectures), for large enough N, TC will be smaller than B.

Figure 1.15 shows that small modifications to the existing intra-block algorithm (Section 1.4.3) allow us to perform a scan on a block of B elements when $TC \leq B$. As we can see, the function `scan_block_anylength` divides the elements into blocks of TC and iterates over the block serially. The key difference is the presence of the variable `reduceValue` that carries the result of the reduce operator on one block of TC elements to the next.

The same kind of modification can be done to the approach shown in Section 1.7.1 to make the intra-block scan function operate on inputs of arbitrary length.

1.8 Optimizations in CUDPP

The code given in Sections 1.4 and 1.5 illustrate the core parallel scan algorithms we use. We make our implementation available as open-source components of CUDPP, the "CUDA Data Parallel Primitives" library (available at `http://www.gpgpu.org/developer/cudpp/`). To achieve peak performance for scan kernels, our CUDPP library combines the basic algorithms described above 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 1.8 and Figure 1.14. 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 that were scanned serially.

```
template<class OP, ScanKind Kind, class T>
__device__ void scan_block_anylength(T *ptr,
                                     const T *in,
                                     T *out,
                                     const uint B,
                                     const uint idx=threadIdx.x,
                                     const uint bidx=blockIdx.x,
                                     const uint TC=blockDim.x)
{
    const uint nPasses = float(ceil(B/float(TC)));

    T reduceValue = OP::identity();

    for (uint i = 0; i < nPasses, ++i)
    {
        const uint offset = i * TC + (bidx * B);

        // Step 1: Read TC elements from global (off-chip)
        // memory to shared memory (on-chip)
        T input = ptr[idx] = in[offset + idx];
        __syncthreads();

        // Step 2: Perform scan on TC elements
        T val = scan_block<OP, Kind, T>(ptr);

        // Step 3: Propagate reduced result from previous block
        // of TC elements
        val = OP::apply(val, reduceValue);

        // Step 4: Write out data to global memory
        out[offset + idx] = val;

        // Step 5: Choose reduced value for next iteration
        if (idx == (TC-1))
        {
            ptr[idx] =
                (Kind == exclusive) ?
                OP::apply(input, val) : val;
        }
        __syncthreads();

        reduceValue = ptr[TC-1];
        __syncthreads();
    }
}
```

FIGURE 1.15: A block level scan that scans a block of B elements with the help of `scan_block`, which does a scan of TC elements at a time, where $TC \leq B$. TC is the number of threads in a block.

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.

1.9 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.

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 single-precision floating point values.

Figure 1.16 compares two scan implementations: our warp-wise scan and the reduce/downsweep algorithm used by Sengupta et al. [17]. Our warp-wise approach is 1 to 20% faster, improving most on scans of non-power-of-two arrays and for arrays smaller than 65K elements. Figure 1.16 compares the reduce/downsweep segmented scan [17] with both of our warp-wise segmented scan kernels: one based on conditional indexing (Fig. 1.11) and one based on operator transformation (Fig. 1.10). 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. [17] 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

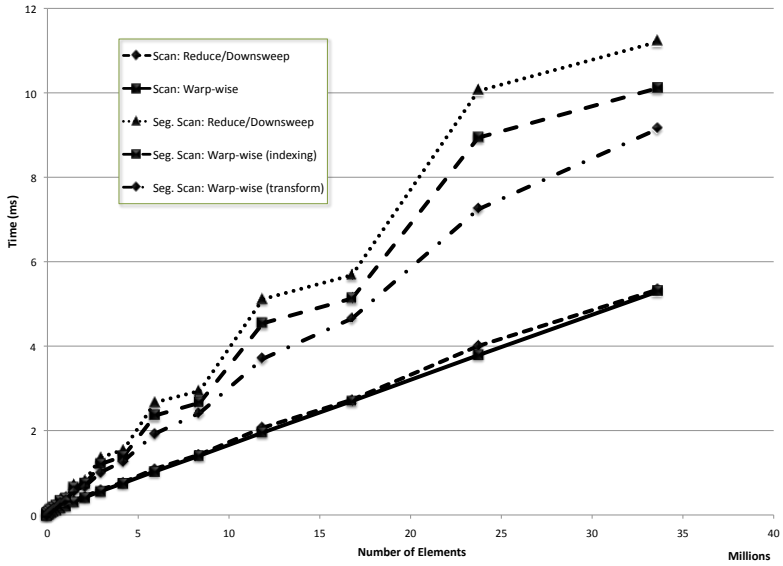

FIGURE 1.16: Scan and Segmented Scan performance.

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 since threads in a warp execute in lock step. 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, 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. Bell and Garland [1] explore this phenomenon in detail using the very important case of sparse-matrix vector multiplication.

Finally, Figure 1.17 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 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

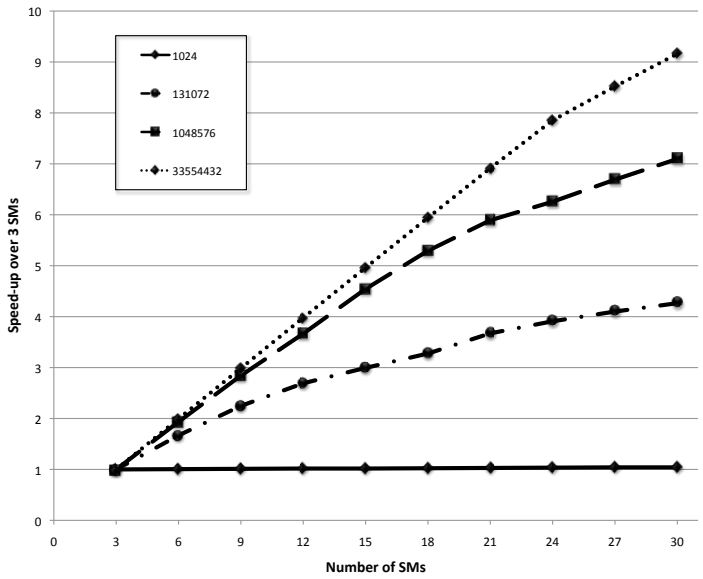

FIGURE 1.17: Parallel scaling of segmented scan on sequences of varying length.

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.

1.10 Conclusions

The modern many-core 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 SIMD 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.

Acknowledgments

Thanks to Dominik Göddeke for his thoughtful comments and suggestions on this chapter. The UC Davis authors thank our research supporters: the DOE/SciDAC Institute for Ultrascale Visualization, the National Science Foundation (award 0541448), and NVIDIA for both a Graduate Fellowship and equipment donations.

Bibliography

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, pages 18:1–18:11, November 2009.

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

[3] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

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

[6] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 666–675, November 1990.

[7] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 205–213. ACM, June 2008.

[8] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Herbert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[9] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[10] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[11] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[12] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.

[13] Aaftab Munshi. *The OpenCL Specification (Version 1.0, Document Revision 48)*, 6 October 2009.

[14] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, March/ April 2008.

[15] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda`, January 2007.

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

[17] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, August 2007.