CHAPTER 3

Optimizing Parallel Prefix Operations for the Fermi Architecture

Mark Harris and Michael Garland

The NVIDIA Fermi GPU architecture introduces new instructions designed to facilitate basic, but important, parallel primitives on per-thread predicates, as well as instructions for manipulating and querying bits within a word. This chapter demonstrates the application of these instructions in the construction of efficient parallel algorithm primitives such as reductions, scans, and segmented scans of binary or Boolean data.

3.1 INTRODUCTION TO PARALLEL PREFIX OPERATIONS

Scan (also known as parallel prefix sums), is a fundamental parallel building block that can form the basis of many efficient parallel algorithms, including sorting, computational geometry algorithms such as quickhull, and graph algorithms such as minimum spanning tree, to name just a few [1]. Given an input sequence of values

$$A = [a_0, a_1, a_2, \dots, a_n]$$

and an arbitrary binary associative operator that we denote $\oplus$, the scan operation produces the output sequence

$$S = scan(\oplus, A) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots, a_0 \oplus \cdots \oplus a_n].$$

This scan is *inclusive* because each element s_i has accumulated values from all elements a_j where $j \le i$. A related operation is the *exclusive* scan, where only elements with $j < i$ are accumulated:

$$X = exclusive\_scan(\oplus, p, A) = [p, p \oplus a_0, p \oplus a_0 \oplus a_1, \dots, p \oplus \cdots \oplus a_{n-1}].$$

The exclusive scan operator requires an additional prefix value p, which is often assumed to be the identity value for the operator $\oplus$.

As a concrete example, applying inclusive scan with the $+$ operator to the input sequence `[3 1 7 0 4 1 6 3]` produces the result `[3 4 11 11 15 16 22 25]`; applying exclusive scan with prefix $p = 0$ to the same input produces the result `[0 3 4 11 11 15 16 22]`.

In this chapter we focus on developing efficient intra-thread-block scan implementations, which are valuable in applications in which scan is used as an operator on local data. When the data are already in registers or on-chip memory, scan is generally instruction limited. A number of recent papers have explored the efficient implementation of global parallel prefix operations in CUDA [2–8]. In most cases, the operators used are simple arithmetic operations, such as integer or floating-point addition. These operators generally map to a single hardware instruction, and therefore the amount of work performed per element of the array is minimal. If the input and output arrays reside in external memory, the cost of transporting the data from external memory to on-chip registers and back dwarfs the cost of computation, and therefore the goal of most efficient GPU scan implementations has been to saturate the available memory bandwidth.

In a number of applications, the input elements are single bits with a value of zero or one. Alternatively, we can consider them to be Boolean predicates. There are two common operations on sequences of elements with corresponding Boolean true/false predicates: *stream compaction* (also known as *filter*), and *split*. Stream compaction removes all elements in the input stream that have false flags [6]. Stream compaction can be used, for example, to take a sequence of object collision test results and generate a sequence of pairs of colliding objects for computing collision response in a physics simulation. The split operation moves all elements with true flags to the beginning of the sequence, and all elements with false flags to the end. We can construct a radix sort for keys of arbitrary length by using the split operation on the sequence of keys by each of their bits in sequence from least to most significant bit [9, 10].

Because of the importance of binary scans, the architects of the NVIDIA Fermi GPU added some new operations for improving their efficiency. These operations allow the threads of a CUDA warp to compute cooperatively. The first is a new `__ballot()` intrinsic function for collecting "votes" from a warp of threads, and the second is a new scalar intrinsic called `__popc()` (short for "population count"). Using these new instructions we can construct efficient binary scans that execute fewer instructions and require much less shared memory than equivalent code that uses full 32-bit operations.

3.1.1 Intra-Warp Scan

Efficient scan implementations on NVIDIA GPUs frequently take advantage of the fact that the graphical processing unit (GPU) executes parallel threads in groups of 32 called warps. The threads of a warp execute in an Single Instruction, Multiple Data (SIMD) fashion, under which they collectively execute each instruction. The CUDA C code in Listing 3.1 shows how one might implement a parallel intra-warp scan primitive.

This function is tailored to operate within a warp in three ways. First, it assumes that the number of threads in a warp is 32, which limits the number of parallel steps to five. Second, because it assumes that the threads of a warp execute synchronously, it omits `__syncthreads()` that would otherwise be necessary when using shared memory. Finally, by padding the intermediate storage for each warp with zeros, it ensures legal indexing of the shared memory array without conditionals (note that this requires extra shared memory). The `s_data` variable is assumed to be a pointer to shared memory, and it must be declared volatile as shown to ensure that the stores to shared memory are executed; otherwise, in the absence of barrier synchronization, the compiler may incorrectly choose to keep intermediate values in registers. Because `s_data` is declared as volatile, it must be reread from shared memory on

```
__device__ int warp_scan(int val, volatile int *s_data)
{
    // pad each warp with zeros
    int idx = 2 * threadIdx.x -
        (threadIdx.x & (warpSize-1));
    s_data[idx] = 0;
    idx += warpSize;
    int t = s_data[idx] = val;

    s_data[idx] = t = t + s_data[idx - 1];
    s_data[idx] = t = t + s_data[idx - 2];
    s_data[idx] = t = t + s_data[idx - 4];
    s_data[idx] = t = t + s_data[idx - 8];
    s_data[idx] = t = t + s_data[idx -16];
    return s_data[idx-1];
}
```

Listing 3.1. CUDA implementation of an intra-warp exclusive plus-scan. Each calling thread passes a single input value to be scanned in `val`, and gets the resulting prefix element in the return value. The pointer `s_data` must point to `__shared__` memory allocated large enough for two int values per calling thread.

every reference; therefore, a separate local variable `t` is required to keep the running sum in a register throughout the scan.

3.2 EFFICIENT BINARY PREFIX OPERATIONS ON FERMI

A central goal of CUDA is to enable cooperation among the threads in a block; this cooperation is very powerful because it enables collective parallel operations that are not bottlenecked by off-chip memory accesses. GPUs with Compute Capability 1.0 implemented the simple `__syncthreads()` barrier synchronization to allow threads within a block to cooperate via shared memory. Compute Capability 1.2 added a warp "vote" instruction that enabled two new intrinsics, `__any()` and `__all()`. Each of these functions takes a predicate[1] as input from each thread that calls it. `__any()` returns true to the calling thread if any thread in the same warp passes a true predicate, and `__all()` returns true only if all threads in the warp pass a true predicate. These intrinsics allow programs to make simple collective decisions within each warp.

Compute Capability 2.0, implemented by the Fermi GPU architecture, provides new instructions that enable more efficient implementation of reduction and scan operations on binary or Boolean inputs. We focus on the case where the operator in question is addition and the values held by each thread are predicates.

[1]We use the term *predicate* interchangeably for values which are (a) a single bit, (b) Boolean true/false values, or (c) integers that are either zero or non-zero.

3.2.1 Primitives Provided by Compute Capability 2.0

The CUDA C compiler provides a number of useful integer arithmetic functions that are supported on all CUDA devices (Appendix C.2.3, [11]). We use the following intrinsic functions to optimize parallel prefix operations.

`int __popc(int x)`	Population Count: Returns the number of bits that are set to 1 in the 32-bit integer `x`.
`int __clz(int x)`	Count Leading Zeros: Returns the number of consecutive zero bits beginning at the most significant bit of the 32-bit integer `x`.

On previous generations of NVIDIA processors, these intrinsic functions mapped to instruction sequences that are tens of instructions long. However, NVIDIA GPUs based on the Fermi architecture (Compute Capability 2.0) provide much more efficient support. In particular, the `__popc()` and `__clz()` intrinsics compile to a single machine instruction.

In addition to providing more efficient support for these integer arithmetic operations, Compute Capability 2.0 GPUs also introduce the capability to efficiently collect predicate values across parallel threads. The first such primitive that we use operates at the warp level.

`int __ballot(int p)`	Returns a 32-bit integer in which bit *k* is set if and only if the predicate `p` provided by the thread in lane *k* of the warp is non-zero.

This primitive is similar in spirit to the existing `__any()` and `__all()` warp vote functions. However, rather than computing a cumulative result, it returns the "ballot" of predicates provided by each thread in the warp. `__ballot()` provides a mechanism for quickly broadcasting one bit per thread among all the threads of a warp without using shared memory.

Finally, the Fermi architecture provides a new class of barrier intrinsics that simultaneously synchronize all the threads of a thread block and perform a reduction across a per-thread predicate. The three barrier intrinsics are:

`int __syncthreads_count(int p)`	Executes a barrier equivalent to `__syncthreads()` and returns the count of non-zero predicates `p`.
`int __syncthreads_and(int p)`	Executes a barrier equivalent to `__syncthreads()` and returns a non-zero value when all predicates `p` are non-zero.
`int __syncthreads_or(int p)`	Executes a barrier equivalent to `__syncthreads()` and returns a non-zero value when at least one predicate `p` is non-zero.

Every thread of the block provides a predicate value as it arrives at the barrier. When the barrier returns, the prescribed value is returned to every thread.

3.2.2 Implementing Binary Prefix Sums with Fermi Intrinsics

In this section we demonstrate the implementation and performance of efficient binary scans and segmented scans using the Compute Capability 2.0 intrinsics described in Section 3.2.1.

3.2.2.1 ***Intra-Warp Binary Prefix Sums***

Consider the case in which every thread of a warp holds a single predicate. By combining the `__ballot()` and `__popc()` intrinsics, we can easily count the number of true values within the warp using the code shown in Listing 3.2.

We call this a binary reduction because the value held by each thread can be represented with a single bit. When the threads of a warp call this function, they each pass a single `bool` flag as an argument to the function. The `__ballot()` intrinsic packs the predicates from all threads in a warp into a single 32-bit integer and returns this integer to every thread. Each thread then individually counts the number of bits that are set in the 32-bit ballot using `__popc()`. This accomplishes a parallel reduction across all threads of the warp.

To compute the prefix sums of predicates across a warp requires only a small modification. Every thread receives the same 32-bit ballot containing the predicate bits from all threads in its warp, but it only counts the predicates from threads that have a lower index. The key is to construct a separate bit mask for each lane of the warp, as illustrated by the code in Listing 3.3.

For each lane k, `lanemask_lt` computes a 32-bit mask whose bits are 1 in positions less than k and 0 everywhere else. To compute the prefix sums, each thread applies its mask to the 32-bit ballot, and then counts the remaining bits using `__popc()`. By modifying the construction of the mask, we can construct the four fundamental prefix sums operations:

1. Forward inclusive scan: set all bits less than or equal to the current lane. (`lanemask_le`)
2. Forward exclusive scan: set all bits less than the current lane. (`lanemask_lt`)

```
__device__ unsigned int warp_count(bool p)
{
    unsigned int b = __ballot(p);
    return __popc(b);
}
```

Listing 3.2. Counting predicates within each warp.

```
__device__ unsigned int lanemask_lt()
{
    const unsigned int lane = threadIdx.x & (warpSize-1);
    return (1 << (lane)) - 1;
}

__device__ unsigned int warp_prefix_sums(bool p)
{
    const unsigned int mask = lanemask_lt();
    unsigned int b = __ballot(p);
    return __popc(b & mask);
}
```

Listing 3.3. Implementation of binary exclusive prefix sums within each warp.

3. Reverse inclusive scan: set all bits greater than or equal to the current lane. (`lanemask_ge`)
4. Reverse exclusive scan: set all bits greater than the current lane. (`lanemask_gt`)

Fermi hardware computes these lane masks internally and provides instructions for reading them, but doing so requires the use of inline PTX assembly. The `lanemask_lt` function in Listing 3.4 produces a lane mask in a single instruction, compared with four for the function in Listing 3.3. The PTX version is also more robust, because it is guaranteed to work correctly for multidimensional thread blocks. The source code accompanying this book includes a header that defines all four versions of `lanemask`.

We can extend the basic binary intra-warp scan (or reduction) technique to scans of multibit integer sequences by performing prefix sums one bit at a time and adding the individual results scaled by the appropriate power of two. For full 32-bit integers, the generic intra-warp scan shown in Listing 3.1 will outperform this approach. The bitwise scan requires one step per bit, but computes 32 results per step, whereas the generic intra-warp scan uses 32-bit arithmetic, but requires five steps to generate 32 results. The cost of one step of bitwise scan is four instructions, and for the generic scan it is three instructions. Clearly the generic scan will be faster for integers with more than about 4 bits, and our experiments confirm this.

3.2.2.2 *Intra-Warp Segmented Binary Prefix Sums*

Segmented scan generalizes scan by simultaneously performing separate parallel scans on arbitrary contiguous segments of the input sequence. For example, applying inclusive scan with the + operator to a sequence of integer sequences gives the following result:

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

Segmented scans are useful in mapping irregular computations such as quicksort and sparse matrix-vector multiplication onto regular parallel execution.

Segmented sequences augment the sequence of input values with a *segment descriptor* that encodes how the sequence is partitioned. A common segment descriptor representation is a *head flags* array that stores 1 for each element that begins a segment and 0 for all others. The head flags representation for the preceding example sequence is:

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

The intra-warp binary scan technique of Section 3.2.2 can be extended to binary segmented scan as demonstrated by the code in Listing 3.5. In addition to the predicate `p`, each thread passes a 32-bit `hd`

```
__device__ unsigned int lanemask_lt()
{
    unsigned int mask;
    asm("mov.u32 %0, %lanemask_lt;" : "=r"(mask));
    return mask;
}
```

Listing 3.4. Efficient lane mask computation using inline PTX assembly.

```
__device__ unsigned int warp_segscan(bool p,
                                     unsigned int hd)
{
    const unsigned int idx = threadIdx.x;
    const unsigned int mask = lanemask_lt();

    // Mask off head flags for lanes above this one.
    // We OR in the 1 because there's an implicit
    // segment boundary at the beginning of the warp.
    hd = (hd | 1) & mask;

    // Count # lanes >= first lane of the current segment
    unsigned int above = __clz(hd) + 1;

    // Mask that is 1 for every lane >= first lane
    // of current segment
    unsigned int segmask = ~((~0U) >> above);

    // Perform the scan
    unsigned int b = __ballot(p);
    return __popc(b & mask & segmask);
}
```

Listing 3.5. Implementation of intra-warp binary segmented prefix sums.

value that contains head flags for its entire warp.[2] `__clz()` is used to count the number of thread lanes greater than or equal to the thread's segment, and the count is used to compute a segment mask that is applied to the ballot along with the lane mask.

3.2.2.3 *Intra-Block Binary Reductions and Prefix Sums*

In CUDA programs, we often need to perform reductions and parallel prefix sums across entire thread blocks. Counting true predicates (a binary reduction) across an entire block simply requires a single call to `__syncthreads_count()`. We compared binary count using `__syncthreads_count()` to an efficient 32-bit parallel reduction [12]. We ran the test on an NVIDIA GeForce GTX 480 GPU using CUDA 3.1 on 32-bit Windows XP, with 512-thread blocks and a wide range of grid sizes. The `__syncthreads_count()` version is on average two times faster, as shown in Figure 3.1, and requires only a single line of code, compared with about 20 lines of code for the 32-bit parallel reduction.[3] `__syncthreads_count()` also uses no shared memory, whereas a 32-bit reduction uses 128 bytes per warp. As in the warp-level case, we can also use `__syncthreads_count()` to sum multibit integers by performing one reduction for each bit of the values, then scaling and adding the results together appropriately.

[2]Note `hd` can be computed from per-thread head flags using `__ballot()`.

[3]Example code is included on the accompanying CD.

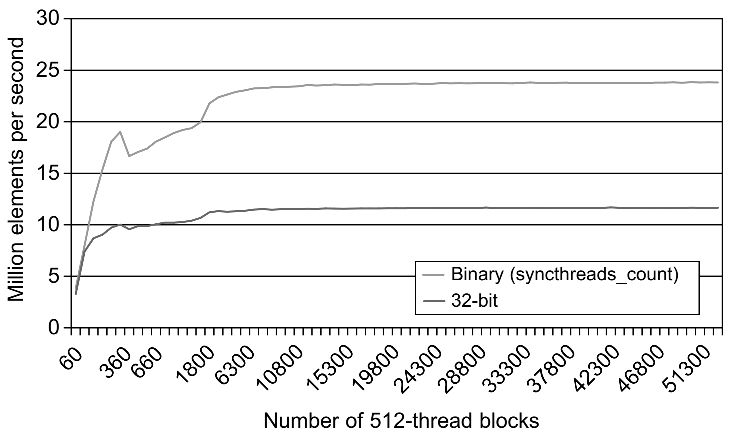

FIGURE 3.1

A performance comparison of intra-block binary counts (reductions) implemented with and without the Fermi `__syncthreads_count()` intrinsic.

```
__device__ int block_binary_prefix_sums(int x)
{
    extern __shared__ int sdata[];
    // A. Compute exclusive prefix sums within each warp
    int warpPrefix = warp_prefix_sums(x);

    int idx = threadIdx.x;
    int warpIdx = idx / warpSize;
    int laneIdx = idx & (warpSize - 1);

    // B. The last thread of each warp stores inclusive
    // prefix sum to the warp's index in shared memory
    if (laneIdx == warpSize - 1)
        sdata[warpIdx] = warpPrefix + x;
    __syncthreads();

    // C. One warp scans the warp partial sums
    if (idx < warpSize)
        sdata[idx] = warp_scan(sdata[idx], sdata);
    __syncthreads();

    // D. Each thread adds prefix sums of warp partial
    // sums to its own intra-warp prefix sums
    return warpPrefix + sdata[warpIdx];
}
```

Listing 3.6. Implementation of intra-block binary exclusive prefix sums.

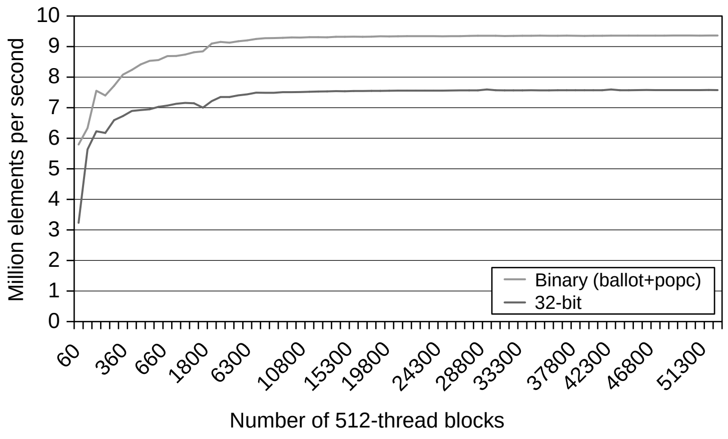

FIGURE 3.2

A performance comparison of intra-block binary prefix sums implemented with and without Fermi `__ballot()` and `__popc()` intrinsics.

In order to compute binary prefix sums across an entire thread block, we use warp-level prefix sums procedures as fundamental building blocks. We perform binary prefix sums within each warp independently, collecting the cumulative sums from each into shared memory. These partial sums are 5-bit integers, which we can combine with a single warp using either 5-bit binary prefix sums or generic 32-bit scan. As discussed in Section 3.2.2, for 5-bit integers, the generic 32-bit scan is slightly faster, so this is what we use in the code in Listing 3.6.

The main optimization that Fermi enables over previous architectures is the use of `__ballot()` and `__popc()` in the `warp_prefix_sums()` function. To measure the benefit of this optimization, we compared the code in Listing 3.6 to the same function modified to call `warp_scan()` rather than `warp_prefix_sums()` in step A. The relative performance of these two versions is shown in Figure 3.2 (The test configuration was the same as used for the binary reduction test of Figure 3.1.) The binary intra-block scan using hardware intrinsics is on average 24% faster than the 32-bit scan version, and the CUDA Visual Profiler shows that it executes about 31% fewer instructions. The binary scan with intrinsics also uses much less shared memory; only the single-warp call to `warp_scan()` in step C of Listing 3.6 requires shared memory, so at most 256 bytes of shared memory are needed per block, compared with 256 bytes *per warp* for a full 32-bit scan.

3.3 CONCLUSION

We have presented binary scan and reduction primitives that exploit new features of the Fermi architecture to accelerate important parallel algorithm building blocks. Often in applications that apply prefix sums to large sequences, bandwidth is the primary bottleneck because data must be shared between thread blocks, requiring global memory traffic and separate kernel launches. We experimented with binary scan and reduction primitives in large binary prefix sums, stream compaction, and radix sort, with average speed-ups of 3%, 2%, and 4%, respectively. For intra-block radix sort, which is not dominated by intra-block communication through global memory, we observed speed-ups as high as 12%.

For applications that are less bandwidth bound, these primitives can help improve binary prefix sums performance by up to 24%, and binary reduction performance by up to 100%. Furthermore, the intrinsics often simplify code, and their lower shared memory usage may help improve GPU occupancy in kernels that combine prefix sums with other computation. Undoubtedly there are further uses for the new instructions provided by Fermi, and we hope that their demonstration here will help others to experiment.

References

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

[2] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3 (Chapter 39), Addison Wesley, New York, 2007, pp. 851–876.

[3] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, Scan primitives for GPU computing, in: Graphics Hardware 2007, 2007, pp. 97–106.

[4] Y. Dotsenko, N.K. Govindaraju, P.-P. Sloan, C. Boyd, J. Manferdelli, Fast scan algorithms on graphics processors, in: Proceedings of the 22nd Annual International Conference on Supercomputing, ACM, 2008, pp. 205–213. http://portal.acm.org/citation.cfm?id=1375527&picked=prox.

[5] S. Sengupta, M. Harris, M. Garland, Efficient Parallel scan algorithms for GPUs, Technical Report NVR-2008-003, NVIDIA Corporation, 2008.

[6] M. Billeter, O. Olsson, U. Assarsson, Efficient stream compaction on wide SIMD many-core architectures, in: HPG ’09: Proceedings of the Conference on High Performance Graphics 2009, ACM, New York, 2009, pp. 159–166.

[7] D. Merrill, A. Grimshaw, Parallel Scan for Stream Architectures, Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009.

[8] S. Sengupta, M. Harris, M. Garland, J.D. Owens, Efficient parallel scan algorithms for many-core GPUs, in: J. Dongarra, D.A. Bader, J. Kurzak (Eds.), Scientific Computing with Multicore and Accelerators, Chapman & Hall/CRC Computational Science (Chapter 19), Taylor & Francis, Boca Raton, FL, 2011, pp. 413–442.

[9] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2009, pp. 1–10.

[10] D. Merrill, A. Grimshaw, Revisiting Sorting for GPGPU Stream Architectures, Technical Report CS2010-03, Department of Computer Science, University of Virginia, 2010.

[11] NVIDIA Corporation, NVIDIA CUDA Programming Guide, version 4.0. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2010 (accessed 27.07.11).

[12] M. Harris, Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/reduction/doc/reduction.pdf, 2007 (accessed 27.07.11).