Parallel Prefix Sum (Scan) with CUDA

Mark Harris
mharris@nvidia.com

April 2007

Document Change History

Version	Date	Responsible	Reason for Change
	February 14, 2007	Mark Harris	Initial release

Abstract

Parallel prefix sum, also known as parallel Scan, is a useful building block for many parallel algorithms including sorting and building data structures. In this document we introduce Scan and describe step-by-step how it can be implemented efficiently in NVIDIA CUDA. We start with a basic naïve algorithm and proceed through more advanced techniques to obtain best performance. We then explain how to scan arrays of arbitrary size that cannot be processed with a single block of threads.

Table of Contents

Introduction

A simple and common parallel algorithm building block is the *all-prefix-sums* operation. In this paper we will define and illustrate the operation, and discuss in detail its efficient implementation on NVIDIA CUDA. As mentioned by Blelloch [1], all-prefix-sums is a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm. The all-prefix-sums operation is defined as follows in [1]:

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

$$[a_0, a_1, \ldots, a_{n-1}],$$

and returns the array

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array

$$[3\ 1\ 7\ 0\ 4\ 1\ 6\ 3],$$

would return

$$[3\ 4\ 11\ 11\ 14\ 16\ 22\ 25].$$

There are many uses for all-prefix-sums, including, but not limited to sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, and building histograms and data structures (graphs, trees, etc.) in parallel. For example applications, we refer the reader to the survey by Blelloch [1].

In general, all-prefix-sums can be used to convert some sequential computations into equivalent, but parallel, computations as shown in Figure 1.

out[0] = 0;
`forall` *j* `from 1 to` *n* `do`
  out[*j*] = *out*[*j*-1] + *f*(*in*[j-1]);
	`forall` *j* `in parallel do`
	  temp[*j*] = *f*(*in*[j]);
	`all_prefix_sums(`*out*`,` *temp*`);`

Figure 1: A sequential computation and its parallel equivalent.

Inclusive and Exclusive Scan

All-prefix-sums on an array of data is commonly known as *scan*. We will use this simpler terminology (which comes from the *APL* programming language [1]) for the remainder of this paper. As shown in the last section, a scan of an array generates a new array where each element *j* is the sum of all elements up to and including *j*. This is an *inclusive scan*. It is often useful for each element *j* in the results of a scan to contain the sum of all previous elements, but not *j* itself. This operation is commonly known as an *exclusive scan* (or *prescan*) [1].

Definition: *The* exclusive scan *operation takes a binary associative operator* $\oplus$ *with identity I*, *and an array of n elements*

$$[a_0, a_1, \ldots, a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

Example: If $\oplus$ is addition, then the exclusive scan operation on the array

[3 1 7 0 4 1 6 3],

returns

[0 3 4 11 11 14 16 22].

An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity. Likewise, an inclusive scan can be generated from an exclusive scan by shifting the resulting array left, and inserting at the end the sum of the last element of the scan and the last element of the input array [1]. For the remainder of this paper we will focus on the implementation of exclusive scan and refer to it simply as *scan* unless otherwise specified.

Sequential Scan

Implementing a sequential version of scan (that could be run in a single thread on a CPU, for example) is trivial. We simply loop over all the elements in the input array and add the value of the previous element of the input array to the sum computed for the previous element of the output array, and write the sum to the current element of the output array.

```
void scan( float* output, float* input, int length)
{
    output[0] = 0; // since this is a prescan, not a scan
    for(int j = 1; j < length; ++j)
    {
        output[j] = input[j-1] + output[j-1];
    }
}
```

This code performs exactly n adds for an array of length n; this is the minimum number of adds required to produce the scanned array. When we develop our parallel version of scan, we would like it to be *work-efficient*. This means do no more addition operations (or *work*) than the sequential version. In other words the two implementations should have the same *work complexity*, $O(n)$.

A Naïve Parallel Scan

```
for d := 1 to log2n do
    forall k in parallel do
        if k ≥ 2^d then x[k] := x[k − 2^(d-1)] + x[k]
```

d:=0 to logn-1
k=0,...,n-1
2^d

Algorithm 1: A sum scan algorithm that is not work-efficient.

The pseudocode in Algorithm 1 shows a naïve parallel scan implementation. This algorithm is based on the scan algorithm presented by Hillis and Steele[1] [4], and demonstrated for GPUs by Horn [5]. The problem with Algorithm 1 is apparent if we examine its work complexity. The algorithm performs $\sum_{d=1}^{\log_2 n} n - 2^{d-1} = O(n \log_2 n)$ addition operations. Remember that a sequential scan performs $O(n)$ adds. Therefore, this naïve implementation is not work-efficient. The factor of $\log_2 n$ can have a large effect on performance. In the case of a scan of 1 million elements, the performance difference between this naïve implementation and a theoretical work-efficient parallel implementation would be almost a factor of 20.

Algorithm 1 assumes that there are as many processors as data elements. On a GPU running CUDA, this is not usually the case. Instead, the `forall` is automatically divided into small parallel batches (called *warps*) that are executed sequentially on a multiprocessor. A G80 GPU executes warps of 32 threads in parallel. Because not all threads run simultaneously for arrays larger than the warp size, the algorithm above will not work because it performs the scan in place on the array. The results of one warp will be overwritten by threads in another warp.

To solve this problem, we need to double-buffer the array we are scanning. We use two temporary arrays (`temp[2][n]`) to do this. Pseudocode for this is given in Algorithm 2, and CUDA C code for the naïve scan is given in Listing 1. Note that this code will run on only a single thread block of the GPU, and so the size of the arrays it can process is limited (to 512 elements on G80 GPUs). Extension of scan to large arrays is discussed later.

```
for d := 1 to log2n do                                          d := 0 to logn-1
    forall k in parallel do
        if k ≥ 2^d then
            x[out][k] := x[in][k − 2^(d-1)] + x[in][k]          2^d
        else
            x[out][k] := x[in][k]
    swap(in, out)
```

Algorithm 2: A double-buffered version of the sum scan from Algorithm 1.

[1] Note that while we call this a *naïve* scan in the context of CUDA and NVIDIA GPUs, it was not necessarily naïve for a Connection Machine [3], which is the machine Hillis and Steele were focused on. Related to work complexity is the concept of *step complexity*, which is the number of steps that the algorithm executes. The Connection Machine was a SIMD machine with many thousands of processors. In the limit where the number of processors equals the number of elements to be scanned, execution time is dominated by step complexity rather than work complexity. Algorithm 1 has a step complexity of $O(\log n)$ compared to the $O(n)$ step complexity of the sequential algorithm, and is therefore step efficient.

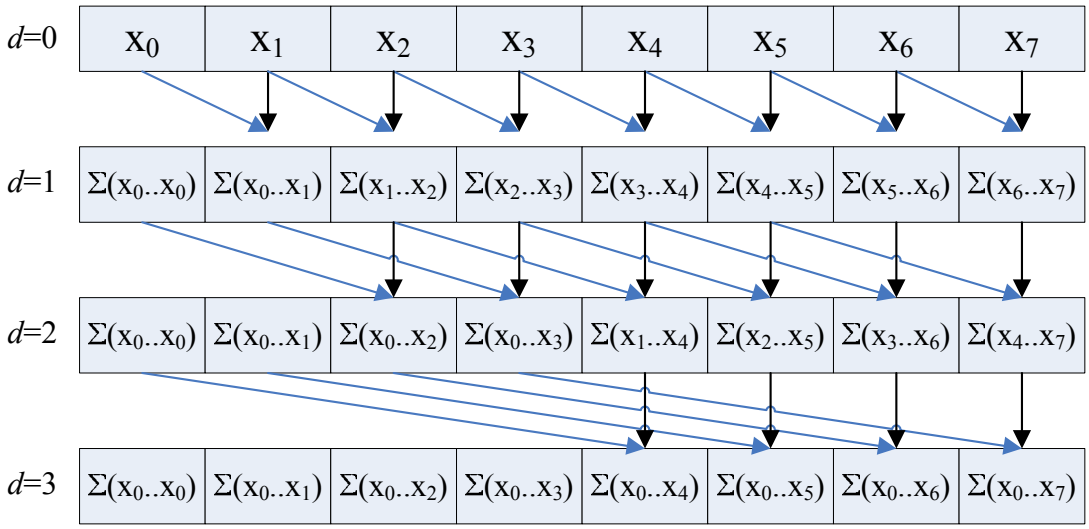

Figure 1: Computing a scan of an array of 8 elements using the naïve scan algorithm.

```
__global__ void scan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // This is exclusive scan, so shift right by one and set first elt to 0
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();

    for (int offset = 1; offset < n; offset *= 2)
    {
        pout = 1 - pout; // swap double buffer indices
        pin  = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads();
    }

    g_odata[thid] = temp[pout*n+thid1]; // write output
}
```

#check the comment by moving the cursor above the green line

Listing 1: CUDA C code for the naive scan algorithm. This version can handle arrays only as large as can be processed by a single thread block running on one multiprocessor of a GPU.

A Work-Efficient Parallel Scan

Our goal in this section is to develop a work-efficient scan algorithm that avoids the extra factor of log n work performed by the naïve algorithm of the previous section. To do this we will use an algorithmic pattern that arises often in parallel computing: *balanced trees*. The idea is to build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum. A binary tree with n leaves has log n levels, and each level $d \in [0,n)$ has 2^d nodes. If we perform one add per node, then we will perform $O(n)$ adds on a single traversal of the tree.

The tree we build is not an actual data structure, but a concept we use to determine what each thread does at each step of the traversal. In this work-efficient scan algorithm, we perform the operations in place on an array in shared memory. The algorithm consists of two phases: the *reduce phase* (also known as the *up-sweep phase*) and the *down-sweep phase*. In the reduce phase we traverse the tree from leaves to root computing partial sums at internal nodes of the tree, as shown in Figure 2. This is also known as a parallel reduction, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array. Pseudocode for the reduce phase is given in Algorithm 3.

In the down-sweep phase, we traverse back up the tree from the root, using the partial sums to build the scan in place on the array using the partial sums computed by the reduce phase. The down-sweep is shown in Figure 3, and pseudocode is given in Algorithm 4. Note that because this is an exclusive scan (i.e. the total sum is not included in the results), between the phases we zero the last element of the array. This zero propagates back to the head of the array during the down-sweep phase. CUDA C Code for the complete algorithm is given in Listing 2. Like the naïve scan code in the previous section, the code in Listing 2 will run on only a single thread block. Because it processes two elements per thread, the maximum array size this code can scan is 1024 elements on G80. Scans of large arrays are discussed later.

This scan algorithm performs $O(n)$ operations (it performs $2*(n-1)$ adds and $n-1$ swaps); therefore it is work efficient and for large arrays, should perform much better than the naïve algorithm from the previous section. Algorithmic efficiency is not enough; we must also use the hardware efficiently. If we examine the operation of this scan on a GPU running CUDA, we will find that it suffers from many shared memory bank conflicts. These hurt the performance of every access to shared memory, and significantly affect overall performance. In the next section we will look at some simple modifications we can make to the memory address computations to recover much of that lost performance.

```
for d := 0 to log2n - 1 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        x[k + 2^(d+1) - 1] := x[k + 2^d - 1] + x [k + 2^(d+1) - 1]
```

Algorithm 3: The up-sweep (reduce) phase of a work-efficient sum scan algorithm (after Blelloch [1]).

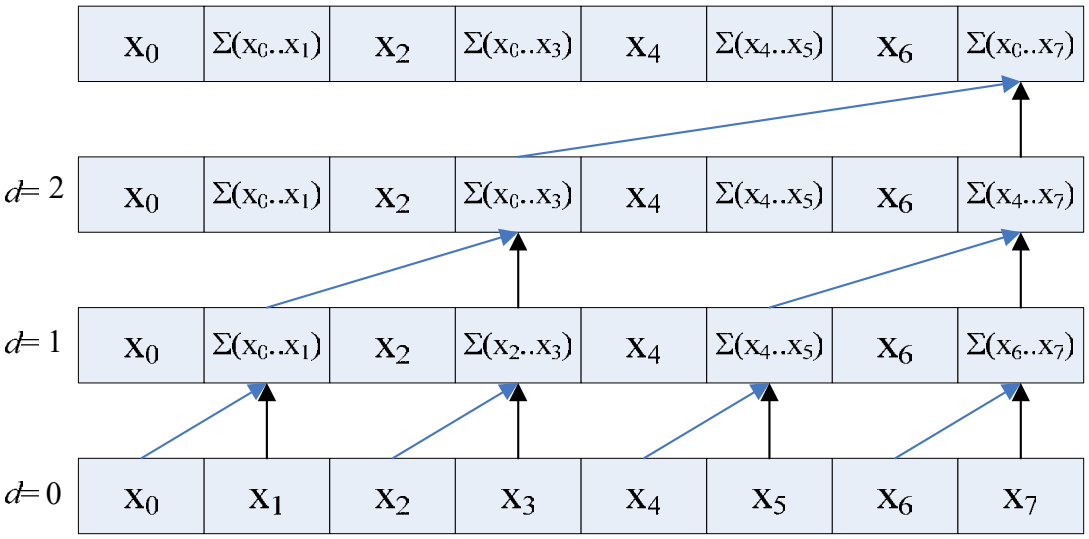

Figure 2: An illustration of the up-sweep, or reduce, phase of a work-efficient sum scan algorithm.

```
x[n - 1] := 0
for d := log2n down to 0 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        t := x[k + 2^d - 1]
        x[k + 2^d - 1] := x [k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] := t + x [k + 2^(d+1) - 1]
```

logn-1

Algorithm 4: The down-sweep phase of a work-efficient parallel sum scan algorithm (after Blelloch [1]).

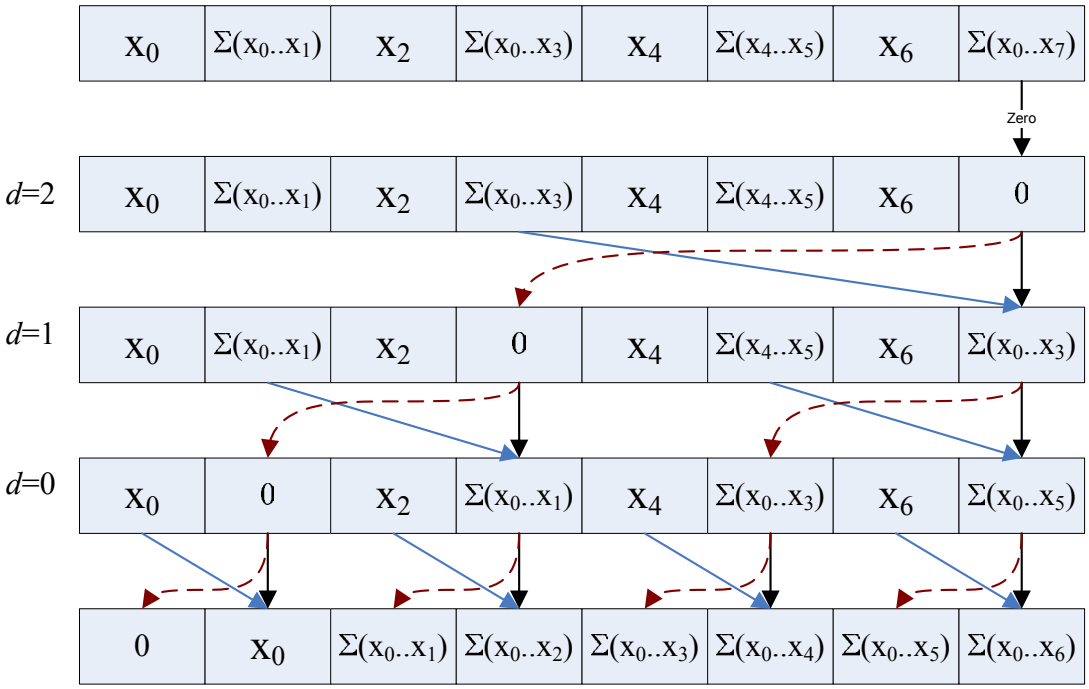

Figure 3: An illustration of the down-sweep phase of the work-efficient parallel sum scan algorithm. Notice that the first step zeros the last element of the array.

per-block prescan: each block contains M threads, which will perform prescan on 2*M elements.

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

A   temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
B           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

C   if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
D           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t   = temp[ai];
            temp[ai]  = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

E   g_odata[2*thid]   = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

Listing 2: CUDA C Code for the work-efficient sum scan of Algorithm 3 and 4. The highlighted blocks are discussed in the next section.

Avoiding Bank Conflicts

The scan algorithm of the previous section performs approximately as much work as an optimal sequential algorithm. Despite this work efficiency, it is not yet efficient on NVIDIA GPU hardware due to its memory access patterns. As described in the NVIDIA CUDA Programming Guide [5], the shared memory exploited by this scan algorithm is made up of multiple banks. When multiple threads in the same warp access the same bank, a bank conflict occurs, unless all threads of the warp access an address within the same 32-bit word. The number of threads that access a single bank is called the *degree* of the bank conflict. Bank conflicts cause serialization of the multiple accesses to the memory bank, so that a shared memory access with a degree-*n* bank conflict requires *n* times as many cycles to process as an access with no conflict. On the G80 GPU, which executes 16 threads in parallel in a half-warp, the worst case is a degree-16 bank conflict.

Binary tree algorithms such as our work-efficient scan double the stride between memory accesses at each level of the tree, simultaneously doubling the number of threads that access the same bank. For deep trees, as we approach the middle levels of the tree the degree of the bank conflicts increases, and then decreases again near the root where the number of active threads decreases (due to the `if` statement in Listing 2). For example if we are scanning a 512-element array, the shared memory reads and writes in the inner loops of Listing 2 experience up to 16-way bank conflicts. This has a significant effect on performance.

Bank conflicts are avoidable in most CUDA computations if care is taken when accessing `__shared__` memory arrays. In convolution, for example, this is just a matter of padding the 2D array to a width that is not evenly divisible by the number of shared memory banks. Scan, due to its balanced-tree approach, requires a slightly more complicated approach. We can avoid most bank conflicts by adding a variable amount of padding to each shared memory array index we compute. Specifically, we add to the index the value of the index divided by the number of shared memory banks. This is demonstrated in Figure 4. We start from the work-efficient scan code in Listing 2, modifying only the highlighted blocks A through E. To simplify the code changes, we define a macro `CONFLICT_FREE_OFFSET`, shown in listing 3.

```
#define NUM_BANKS 16
#define LOG_NUM_BANKS 4

#ifdef ZERO_BANK_CONFLICTS
#define CONFLICT_FREE_OFFSET(n) \
    ((n) >> NUM_BANKS + (n) >> (2 * LOG_NUM_BANKS))
#else
#define CONFLICT_FREE_OFFSET(n) ((n) >> LOG_NUM_BANKS)
#endif
```

Listing 3: This macro is used for computing bank-conflict-free shared memory array indices.

The blocks A through E in Listing 2 need to be modified using these macros to avoid bank conflicts. Two changes must be made to block A. Each thread loads two array elements from the `__device__` array `g_idata` into the `__shared__` array `temp`. In the original code, each thread loads two adjacent elements, resulting in interleaved indexing of the shared memory array, incurring two-way bank conflicts. By instead loading two elements from separate halves of the array, we avoid these bank conflicts. Also, to avoid bank conflicts during the tree traversal, we need to add padding to the shared memory array every `NUM_BANKS` (16) elements. We do this using the macros in Listing 3 as in the following code blocks A through E. Note that we store the offsets to the shared memory indices so that we can use them again at the end of the scan when writing the results back to the output array `g_odata` in block E.

Block A:

```
int ai = thid;
int bi = thid + (n/2);

int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
int bankOffsetB = CONFLICT_FREE_OFFSET(ai);

temp[ai + bankOffsetA] = g_idata[ai];
temp[bi + bankOffsetB] = g_idata[bi];
```

Blocks B and D are identical:

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += CONFLICT_FREE_OFFSET(ai);
bi += CONFLICT_FREE_OFFSET(bi);
```

Block C:

```
if (thid==0) { temp[n – 1 + CONFLICT_FREE_OFFSET(n - 1)] = 0; }
```

Block E:

```
g_odata[ai] = temp[ai + bankOffsetA];
g_odata[bi] = temp[bi + bankOffsetB];
```

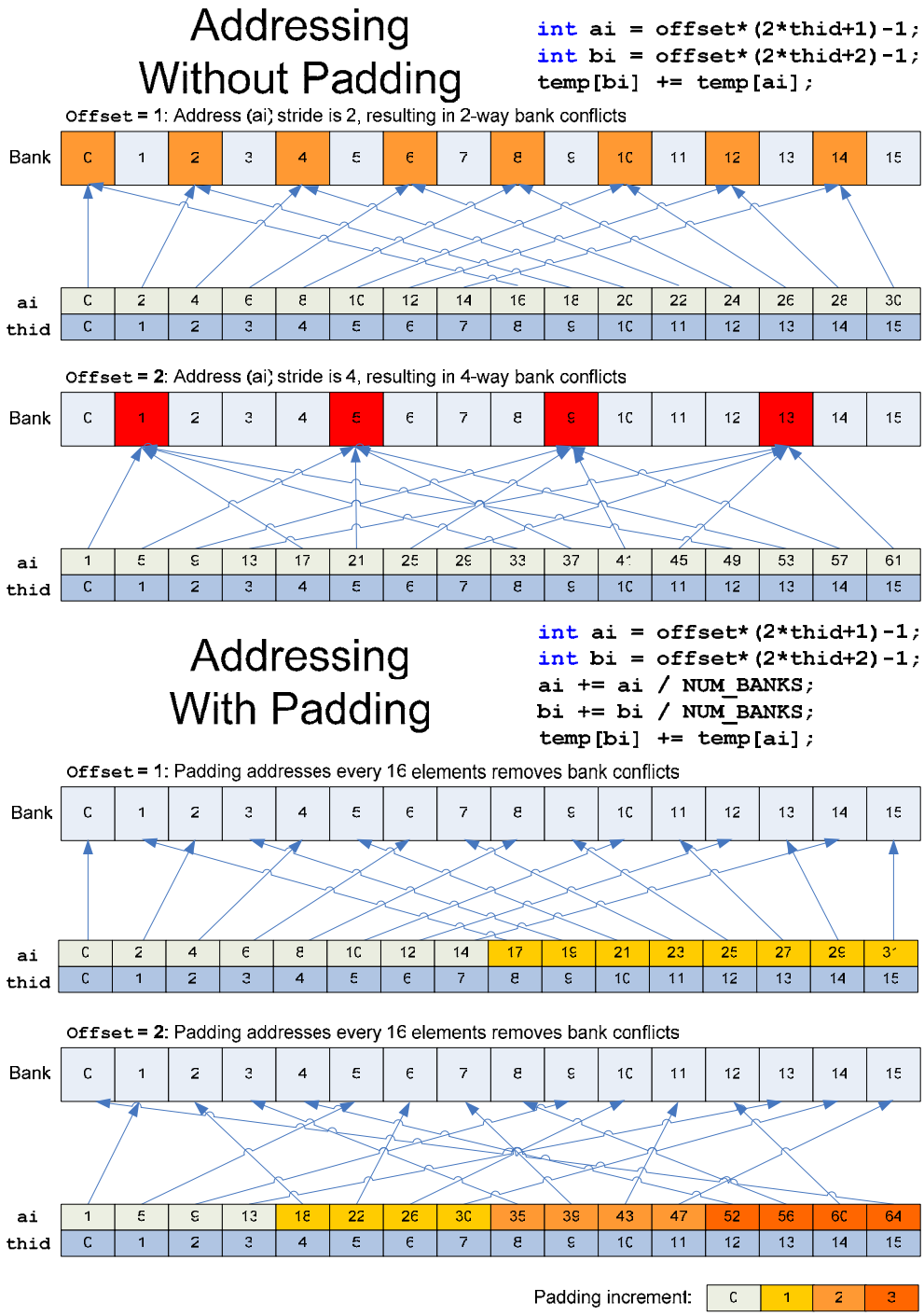

Figure 4: Simple padding applied to shared memory addresses can eliminate high-degree bank conflicts during tree-based algorithms like scan. The top of the diagram shows addressing without padding and the resulting bank conflicts. The bottom shows padded addressing with zero bank conflicts.

Arrays of Arbitrary Size

The algorithms given in the previous sections scan an array inside a single thread block. This is fine for small arrays, up to twice the maximum number of threads in a block (since each thread loads and processes two elements). On G80 GPUs, this limits us to a maximum of 1024 elements. Also, the array size must be a power of two. In this section we explain how to extend the algorithm to scan large arrays of arbitrary (non-power-of-two) dimensions. This algorithm is based on the explanation provided by Blelloch [1].

The basic idea is simple. We divide the large array into blocks that each can be scanned by a single thread block, scan the blocks, and write the total sum of each block to another array of block sums. We then scan the block sums, generating an array of block increments that that are added to all elements in their respective blocks. In more detail, let N be the number of elements in the input array *IN*, and B be the number of elements processed in a block. We allocate N/B thread blocks of $B/2$ threads each (In this section we assume that N is a multiple of B, and extend to arbitrary dimensions in the next paragraph). A typical choice for B on G80 GPUs is 512. We use the scan algorithm of the previous sections to scan each block *j* independently, storing the resulting scans to sequential locations of an output array *OUT*. We make one minor modification to the scan algorithm. Before zeroing the last element of block *j* (label B in Listing 2), we store the value (the total sum of block *j*) to an auxiliary array *SUMS*. We then scan *SUMS* in the same manner, writing the result to an array *INCR*. We then add *INCR(j)* to all elements of block *j* using a simple uniform add kernel invoked on N / B thread blocks of $B / 2$ threads each. This is demonstrated in Figure 4. For details of the implementation, please see the source code for the sample "`scan_largearray`" in the NVIDIA CUDA SDK.

To handle non-power-of-two dimensions, we simply divide the array into a part that is a multiple of B elements and process it as above (using $B/2$ threads per block), and process the remainder with a scan kernel modified to handle non-power-of-2 arrays in a single block. This kernel pads the shared memory array used out to the next higher power of two and initializes this extra memory to zero while loading in the data from device memory. For details see the source code for the sample "`scan_largearray`".

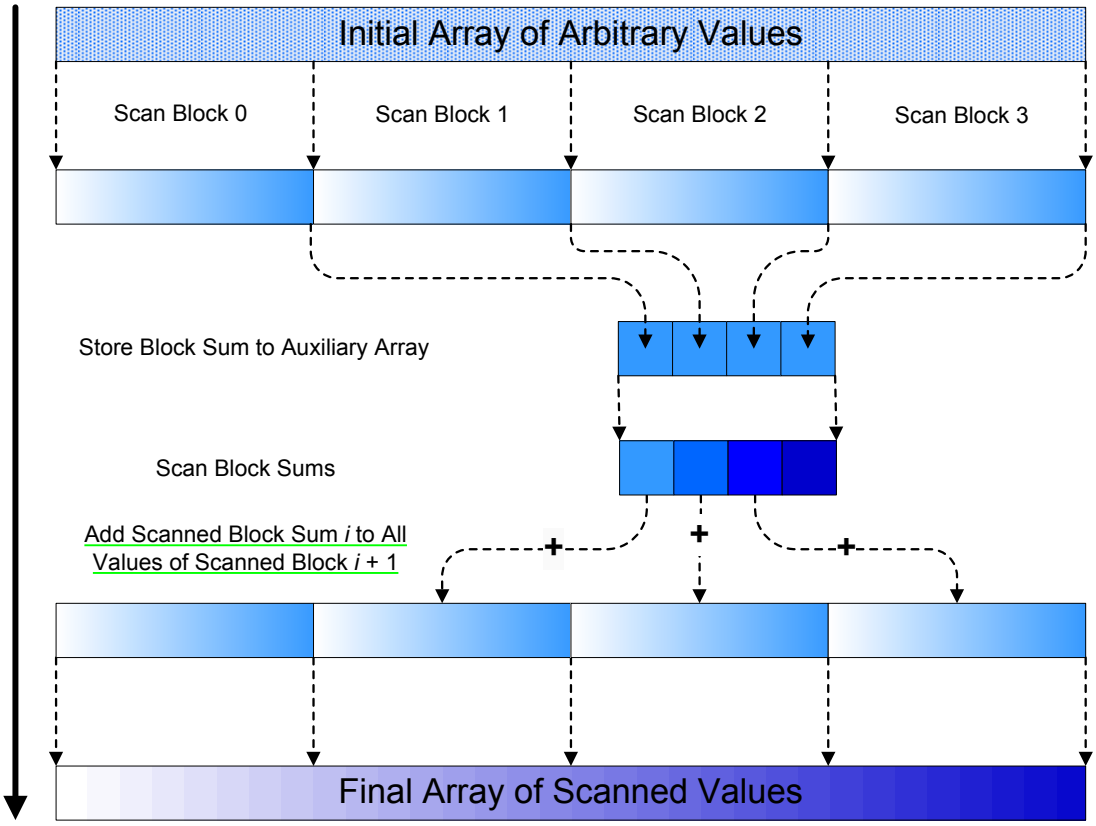

Figure 5: Algorithm for performing a sum scan on a large array of values.

Performance

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Table 2: Performance of the work-efficient, bank conflict free Scan implemented in CUDA compared to a sequential scan implemented in C++. The CUDA scan was executed on an NVIDIA GeForce 8800 GTX GPU, the sequential scan on a single core of an Intel Core Duo Extreme 2.93 GHz.

Conclusion

The scan operation is a simple and powerful parallel primitive with a broad range of applications. In this technical report we have explained the efficient implementation of scan using CUDA which achieves a significant speedup over a sequential implementation on a fast CPU. In the future, we will add example applications of scan such as sorting and stream compaction.

Bibliography

1. Guy E. Blelloch. "Prefix Sums and Their Applications". In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1990. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-90-190.html
2. Siddhartha Chatterjee and Jan Prins. "COMP 203: Parallel and Distributed Computing. PRAM Algorithms". Course Notes. Fall 2005. http://www.cs.unc.edu/~prins/Classes/203/Handouts/pram.pdf
3. Hillis, W. Daniel. *The Connection Machine*. The MIT Press, 1985.
4. Hillis, W. Daniel, and Steele Jr., Guy L. Data Parallel Algorithms. *Communications of the ACM* 29, 12, pp. 1170–1183. ACM Press, December 1986. http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=7903
5. Horn, Daniel. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, Ed., ch. 36, pp. 573–589. Addison Wesley, 2005 http://developer.nvidia.com/object/gpu_gems_2_home.html
6. NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. 2007.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com