StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization

Shengen Yan[1,2,3] Guoping Long[1] Yunquan Zhang[1,2]

[1]Lab. of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences
[2]State Key Laboratory of Computing Science, the Chinese Academy of Sciences
[3]Graduate University of Chinese Academy of Sciences

yanshengen@gmail.com, guoping@iscas.ac.cn, zyq@mail.rdcps.ac.cn

Abstract

Scan (also known as prefix sum) is a very useful primitive for various important parallel algorithms, such as sort, BFS, SpMV, compaction and so on. Current state of the art of GPU based scan implementation consists of three consecutive Reduce-Scan-Scan phases. This approach requires at least two global barriers and 3N (N is the problem size) global memory accesses. In this paper we propose StreamScan, a novel approach to implement scan on GPUs with only one computation phase. The main idea is to restrict synchronization to only adjacent workgroups, and thereby eliminating global barrier synchronization completely. The new approach requires only 2N global memory accesses and just one kernel invocation. On top of this we propose two important optimizations to further boost performance speedups, namely thread grouping to eliminate unnecessary local barriers, and register optimization to expand the on chip problem size. We designed an auto-tuning framework to search the parameter space automatically to generate highly optimized codes for both AMD and Nvidia GPUs. We implemented our technique with OpenCL. Compared with previous fast scan implementations, experimental results not only show promising performance speedups, but also reveal dramatic different optimization tradeoffs between Nvidia and AMD GPU platforms.

Categories and Subject Descriptors D.1.3 [**Concurrent Programming**]: Parallel programming

Keywords Scan, prefix-sum, OpenCL, CUDA, GPU, Parallel algorithms

1. Introduction

Scan, also known as prefix-sum, is a very important problem in parallel computing. Blelloch [1] first introduced scan as a fundamental primitive and discussed its possible applications [2].Later on, more and more parallel applications of scan emerged, such as sort [6, 7, 8, 9, 10, 11], BFS [12, 13], SpMV [13, 14], parallel compaction [15],minimal spanning tree [16] and linked list prefix computations [17], etc. The (inclusive) scan problem is defined as follows:

Given a sequence with n input elements:

$$[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$$

Return a sequence with n output elements:

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2} \oplus a_{n-1})]$$

The $\oplus$ symbol denotes a binary reduction operator, which satisfies the associative law and commutative law. The sequential algorithm to compute prefix sums is simple. Over the years several well known parallel scan algorithms have been proposed [3, 4, 5]. However, it is still surprisingly challenging for high performance parallel implementation, especially when considering target hardware architecture specifics.

In recent years, GPUs emerge as a promising platform for various computation intensive applications. At the same time a lot of optimization skills proposed on GPUs [29]. Compared with CPUs, GPUs have much higher computation density and off chip memory bandwidth. For both Nvidia and AMD GPUs, the ultra high computing power comes from many processing cores organized in a highly hierarchical manner. On the high level, although scan has relatively low computation density, there is ample inherent parallelism. This, combined with the high memory bandwidth of GPUs, makes it possible to achieve promising speedups. Hence, over the years many fast scan algorithms [20, 21, 24, 28] have been proposed on GPU platforms.

Current state of the art scan algorithms [20, 21] on GPUs consist of three Reduce-Scan-Scan phases. The idea is to partition the computation into three consecutive steps. Threads of all workgroups work together to complete each step one by one. Therefore, global barrier synchronization is needed between successive steps. Because of this, this approach requires at least 3N (N is the problem size) global memory accesses, which is detrimental to scan like memory intensive algorithms.

At the micro-architecture level, there are two major flavors of designs for the instruction pipelines. One is the scalar approach on Nvidia GPUs. The other is VLIW on AMD GPUs. To the best of our knowledge, current state of the art implementations only explored the optimization space for scalar architectures, neglecting the fact that the same set of optimization techniques may have drastic different implications on different hardware platforms.

In this paper, we propose StreamScan, a novel scan algorithm for GPUs. Compared to current state of the art, StreamScan has two unique features. First, by restricting synchronization only to adjacent workgroups, we eliminate global barriers completely (Section 3.1). This reduces global memory traffic from at least 3N to only ~2N. On top of this we propose two important optimizations to further improve performance, namely thread grouping to eliminate unnecessary local barriers, and register optimization to expanding the on chip problem size (Section 3.3). Second, we explore the optimization space across both Nvidia and AMD

PPoPP'13, February 23–27, 2013, Shenzhen, China.

GPUs. To facilitate this task, we implement our algorithm with OpenCL. Our experimental results show not only superior performance speedups, but also reveal very different optimization tradeoffs on both platforms (Section 4.2).

Specifically, we make the following contributions:

a) We propose StreamScan, a novel scan algorithm on GPUs which requires only ~2N global memory accesses;
b) We propose thread grouping, an optimization to eliminate redundant local barriers while loading data to local memory. Instead of arranging local memory data as a 2D matrix as MatrixScan [21], we organize the data in 3D manner.
c) We propose register optimization to exploit register file space to enlarge the solvable problem size on chip.
d) We propose a performance auto-tuning framework to explore the parameter space for optimal implementations for both Nvidia and AMD GPUs automatically.

The terminology used in our work is based on the OpenCL specification [30]. This paper is organized as follows. Section 2 discusses research background and motivations of our work. Section 3 presents the StreamScan algorithm. Section 4 presents experimental results and our main findings. Section 5 discusses related works. The final section concludes our work.

2. Background & Motivation

Parallel scan algorithms on CPU architectures are well known [1, 2]. Compared to CPUs, GPUs have much higher computation density and off-chip memory bandwidth. Therefore there is strong incentive to parallelize scan on GPU platforms. However, because of the complex, deep memory hierarchies and week memory consistency models, high performance parallel scan implementations on GPUs are much more challenging. In this section, we review current state of the art of parallel scan algorithms on GPUs and motivate our approach.

2.1 Existing scan algorithms on GPUs

Existing fast scan implementations on GPUs all exploit the hierarchical parallelism of the algorithm. On the high level, the workspace is partitioned into multiple workgroups (or blocks) of workitems. The design space is twofold. The first is the design of an inter-block orchestration mechanism. The second is intra-block scan implementation.

2.1.1 Inter-block orchestration mechanism

An intuitive, yet memory consuming way to perform scan is a three phase Scan-Scan-Add approach[23], as shown in Figure 1. In the first phase, the input array is partitioned into a number of blocks and each block is scanned locally. After this, the last element of each block is stored into an intermediate array *I* according to the block order. The second phase is to perform exclusive scan on the intermediate array *I*. In the last phase, for the *k-th* block (except the first block), each element within the block is added by *I[k]* of the intermediate array. Global barriers are needed between successive phases. This approach requires at least 4N global memory accesses. If the barrier is implemented by a kernel invocation, it also suffers from kernel invocation overheads.

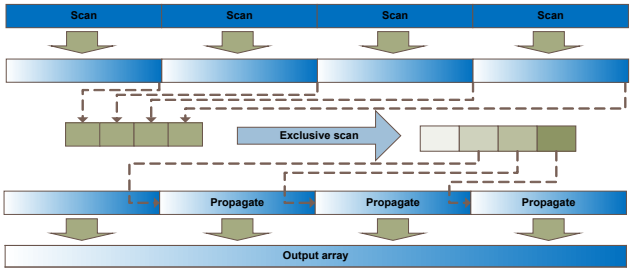


Figure 1. Scan-Scan-Add inter-block orchestration mechanism.

A better three-phase approach is the Reduce-Scan-Scan [21], as shown in Figure 2. In the first phase, the input array is partitioned into multiple blocks, and each block is reduced rather than scanned. The result of each block is stored in the intermediate array *I*, according to the block order. The second phase is the same, to perform exclusive scan on the intermediate array *I*. In the last phase, each block is scanned. After this, for the *k-th* block (except the first block), each element within the block is added by *I[k]*. By moving the block local scan from the first phase to the third phase, the number of global accesses is reduced from 4N to 3N.

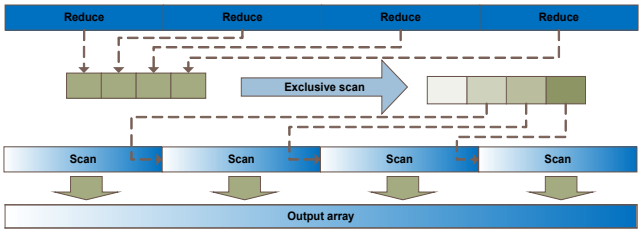


Figure 2. Reduce-scan-Scan inter-block orchestration mechanism.

The size of the intermediate array depends on the number of partitions of the input array. In the original implementation [21], the size can be so large that it is impossible to perform scan on intermediate array with on workgroup. In this case recursive partitioning of the array is needed, and thus can incur further implementation complexity and performance overheads. This issue can be addressed by fixing the size of the intermediate array by using static threads [24].

2.1.2 Intra-block scan implementation

One common intra-block scan implementation is the tree based approach, which realizes a well known two-phase work-efficient algorithm [25]. On GPU platforms, the challenge is load balance and potential bank conflicts (if on-chip local memory (shared memory in CUDA) is used for data buffering).

A better approach for intra-block implementation is MatrixScan [21], which is in essence another three phase Reduce-Scan-Scan in local memory, as shown in Figure 3. The input 1D array is arranged as a 2D matrix in local memory. The number of rows is the same as the number of threads in a workgroup. In the first phase, each thread performs reduce on a row in the matrix sequentially and stores results into an auxiliary array *C*. A local barrier is necessary to ensure that all threads have finished before proceed to the next phase. In the second phase, the auxiliary array *C* is scanned using several threads for best performance. Another local

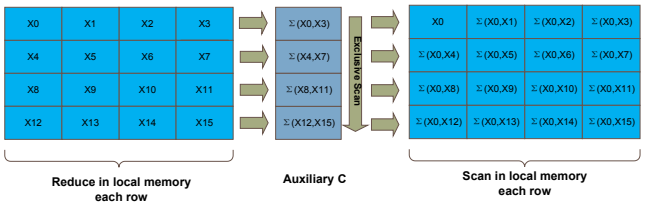


Figure 3. 2D Matrix Scan.

Table 1. Comparison of Our Work to Previous Works.

Strategy	Global memory accesses	Global Barriers	Cross Platform Analysis
Scan-Scan-Add[23]	~4N	>=2	No
Reduce-Scan-Scan[20]	~3N	>=2	No
StreamScan	**~2N**	**None**	**Yes**

barrier is needed here. Finally in the last phase, each thread scans its corresponding row again taking the corresponding results of array *C* to the first element. Potential local memory bank conflicts can be reduced substantially by appropriate column padding to the matrix.

2.2 Motivation of this work

All fast scan implementations on GPUs partition the input array into multiple blocks, and perform computation and synchronization at block granularity. However, the main issue with all variants of the three-phase approach is the global barrier synchronization. Because of the limitation of the on-chip memory, global barrier will suffer global memory traffic overheads. If the global barrier is implemented via kernel invocations, there are also extra runtime overheads.

There is another perspective to understand the scan problem, especially when the problem size is large. Instead of partitioning the computation horizontally into three phases, it's possible to partition the data into multiple blocks, and perform scan at block granularity sequentially. Although the parallelism at the block level is decreased (there is data dependence between adjacent blocks), there is ample parallelism while scanning within the block. In this case, synchronization occurs only between adjacent blocks, global barriers are not necessary. In this algorithm, we need to cache the processing data in the on-chip memory until the whole work completed in the same block. However, we only need to invoke one kernel and since the early exit workgroups will free the on-chip resources, the number of global memory accesses can be reduced to approximate 2N. There are two major challenges to this approach. First, how to fully exploit the parallel computation power of the GPU while scanning one block? Second, how to determine the optimal block size in order to balance between synchronization overheads and kernel performance?

The design space of the intra-block scan implementation also deserves detailed exploration. First, is the traditional intra-block scan approach still suitable for the StreamScan? Second, since the algorithm is memory intensive, how to access the global memory efficiently? Third, due to the large register file size of modern GPUs, how to fully leverage its space to increase the solvable problem size on-chip for the scan problem? Fourth, optimal implementations on different GPUs depend on a set of key parameters. An automatic parameter searching and tuning framework is necessary to generate optimal code versions for various GPU platforms. Table1 summarizes key features of our approach. We will all challenges in detail in the next section.

3. StreamScan

In this section, we present StreamScan, our approach for fast scan implementation on GPUs. We first review StreamScan architecture, and especially how this organization can fully exploits the parallel computation capability of modern GPUs. We then discuss the synchronization mechanism between adjacent blocks. In Section 3.3 we present various intra-block scan algorithms within the StreamScan framework.

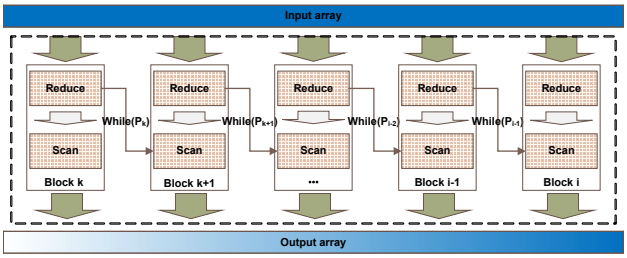


Figure 4. StreamScan Architecture (P is the sync condition).

3.1 Inter-Block orchestration

Figure 4 illustrates the high level architecture of StreamScan. The input data array is partitioned into multiple blocks. Then each block is scanned with a three-phase Reduce-Scan-Scan intra-block algorithm. After all blocks have been scanned, results are written to the output array.

However, there are data dependences between adjacent blocks. That is, scan of block *i* must take the accumulated result of all previous blocks as input. Luckily this accumulated result can be generated after the first reduce phase of the intra-block scan. Thus inter-block synchronization happens only after the first phase. Note that the first reduce phases of all blocks can proceed simultaneously. Abundant inter-block parallelism is available.

Modern GPUs have a hierarchy of parallel processing units. For AMD GPUs, one chip has multiple compute units (CU), with each CU containing a number of processing cores. For StreamScan to work on practical problem sizes, typically the number of blocks is much larger than the number of CUs. We therefore map all intra-block scans on blocks to all CUs. Since adjacent block synchronization occurs only after the first phase, there is still ample parallelism to keep all CUs busy.

The mapping of intra-block scan to all cores within the compute unit is critical to performance. In order to eliminate global barriers completely, we assign one workgroup for each block scan. Parallelism in all three phases is mapped uniformly among all cores. The thread organization within the workgroup is tuned automatically to ensure proper computation overlap and latency hiding. Design details are in Section 3.3. Note that the parallelization and computation mapping framework can be applied to Nvidia GPUs in a similar way.

3.2 Block synchronization

Inter-block synchronization is performed via shared global memory. Like previous algorithms [20], we also need an intermediate array *I* to store reduction results of all workgroups. The difference is that in our approach, the array *I* not only stores reduction results but also serves as a flag variable array for adjacent block synchronization. At the beginning, we initialize all elements of the flag array to a fixed value *F*. Since we can't guarantee the execution order of different workgroups, we initialize the array on the host side before upload it to GPU memory. Compared to the input array to be scanned, the size of the intermediate array is very small (~2K elements for the input array with 16M elements). Thus the upload overhead of the initial intermediate array is negligible.

```
gid: workgroup id.
lid: thread local id.
I: intermediate array.
r:The reduction value of current workgroup;
s: The reduction value of the previous workgroups.
F: The initial value of the intermediate array I.
void ADJACENT_SYNCHRONIZATION(int gid, int lid, __global volatile int *I, int r, __local int *s)
{
    if(lid==0) {
        int p=0;
        if(gid == 0) I[0] = r;
        else{
          while((p=I[gid - 1])== F){}
          I[gid] = p + r;
        }
        *s = p;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Figure 5. Adjacent synchronization.

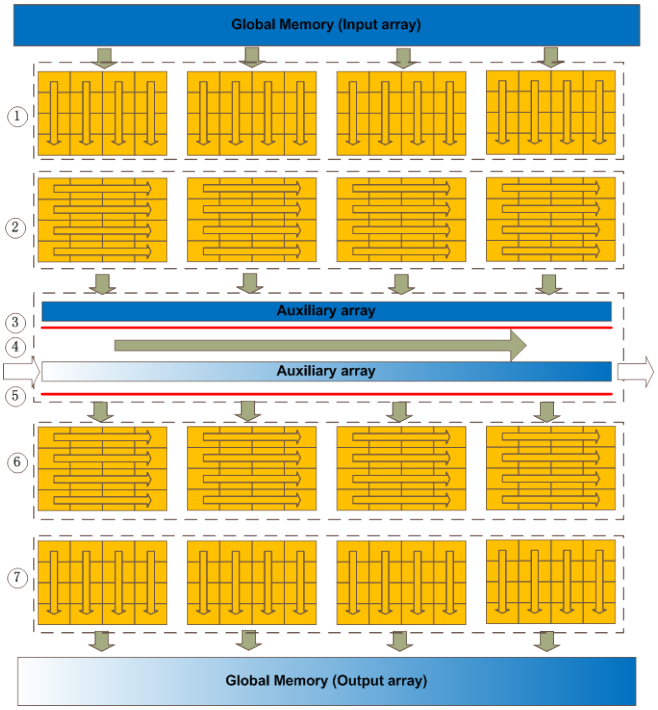
The inter-block synchronization algorithm is shown in Figure 5. For each workgroup, only the first thread (lid=0) is necessary to execute the while loop. A local barrier is needed to make sure that only after the first thread has obtained the reduction result of the previous workgroup, other threads in the same workgroup can continue execution. In order to broadcast the reduction value of the previous workgroup to all threads in the workgroup, we use the on-chip local memory (shared memory in CUDA) to store the reduction result. The purpose of the ***volatile*** keyword is to prevent the compiler from caching stale results to the register file. This guarantees that each workgroup can get the latest value from the intermediate array once the previous workgroup has produced the result.

This algorithm works fine when the on-chip cache capability is disabled. In some platforms, the cache is enabled by default. If the cache between the register file and global memory is enabled, it's still possible to get stale cached copies in each iteration of the while loop. One possible way to address this issue is using atomic primitives to fetch the reduction value of the previous workgroup.

3.3 Intra-Block scan algorithm

In principle, any local scan algorithm can fit into the StreamScan framework. There are three basic approaches for our purpose. A simple one is the classical tree based scan [25], but suffers from the issues like load imbalance among all threads and potential bank conflicts if on-chip local memory is used as data buffers.

A better approach is the three-phase Scan-Scan-Add algorithm. Although it does not suffer from issues faced by the tree based scan, there are two other issues, which may throttle its efficiency. As discussed in Section 3.1, each workgroup (except the starting workgroup) synchronizes with its previous one to retrieve the reduction result at the end of the first phase. While the first scan phase can generate a reduction result, it is much more time consuming than a direct reduction phase. The result is that the delayed inter-block synchronization delays the freedom of dynamic



Steps of 3D MatrixScan. ① Loads data from global memory to local memory. The arrow direction shows the access order of a thread. ② Performs reduction on the row of the matrix. Each thread is in charge of a row. ③ Local Barrier, which ensures all threads have written their reduction results to the auxiliary array. ④ Scan of the on-chip auxiliary array and add the accumulation value of the previous workgroups. The reduction result is sent to the next workgroup via adjacent synchronization. ⑤ Local Barrier, which ensures the scan on auxiliary array is finished. ⑥ Propagates the auxiliary array to the first element of each row in the matrix, and then performs scan on each row of the matrix. ⑦ Store scanned results back to global memory

Figure 6. 3D MatrixScan.

parallelism. Another issue is the on-chip local memory traffic. The data array needs to be updated in place in both the first scan and the third add phases.

The third approach is the Reduce-Scan-Scan algorithm. Compared with Scan-Scan-Add, since Reduce is much faster than Scan, adjacent block synchronization can complete as early as possible. Moreover in the first phase, only the reduction result is needed. It's not necessary to update any element of the data array in local memory. We therefore base our intra-block scan implementation on the Reduce-Scan-Scan algorithm.

An efficient way to implement the three-phase scan algorithm is matrix based scan [21]. In StreamScan, the matrix based intra-block scan consists of three basic steps. In the first step, the input data is loaded from global memory to on-chip space (local memory and registers) and organized as a multidimensional matrix. Next we perform reduction on each row of the matrix and store the reduction result to an auxiliary array and perform a scan on the auxiliary array. In the second step, we propagate the reduction value of current workgroup to the next one and each workgroup except the first one synchronizes with its previous workgroup to retrieve the reduction result. In the last step, we perform a scan on each row after propagating the accumulation result of the previous rows and the previous workgroups to the first elements in the row.

The primary challenge with intra-block scan is how to balance three performance critical tradeoffs. This includes: (1) Optimal

matrix dimension, 2D or 3D? (2) The tradeoff between memory efficiency and implementation complexity; (3) What's the sweet spot by using register space to enlarge the solvable on chip problem size? In the rest of the subsection we discuss each tradeoff in detail.

3.3.1 2D Matrix VS. 3D Matrix

In order to coalesce off chip memory accesses as much as possible, the traditional 2D matrix based intra-block scan will load the data in the local memory as a transposed manner compared with storing them. This discrepancy of access ordering necessitates a local barrier between loading data from global memory and actual computation. This hurts performance, especially so when we use the register file space to enlarge the on chip working set.

Instead of 2D matrix arrangement, we propose a novel organization to eliminate unnecessary local barriers, as shown in Figure 6. We call this technique thread grouping in this paper. The basic idea is to arrange threads of the same workgroup into several no more than wavefront (warp) size groups. Each group loads and processes a sub-block of the input array independently, following the 2D matrix arrangement approach. By treating the thread group as a third dimension, we transform the input array into a logical 3D matrix. For each group, in order to coalesce off chip memory accesses as much as possible, there still exist access transpose of the matrix. However, no explicit local barrier is needed because the hardware will ensure that all threads within the same wavefront (warp) will complete loading data into local memory in a synchronized manner. This is important if we use the register to further enlarge the on chip working set (section 3.3.4).

In our 3D arrangement, each group is responsible for a 2D array. Each thread within the group processes a row. This access pattern is prone to bank conflicts. By appropriately padding extra columns, the number of bank conflicts can be reduced quit a lot.

3.3.2 Expanding working set via register space

On modern GPUs, the register file size is much larger than the local memory size. One advantage of large register file size is to hold as many thread contexts on chip as possible. While more thread contexts may enable more potential for latency hiding, it also increases the thread management overhead. A balanced implementation achieves latency hiding with a reasonable number of threads. In this case, it makes sense to expand the solvable problem size on chip by exploiting register storage.

As discussed in the previous section, we partition all threads within the workgroup into multiple groups. Each group processes a data segment arranged as a 2D array. This array is a logical structure, because physically we partition it into two components: one in the register file, and the other in local memory.

With this partitioned array structure, an immediate issue is how to load data into the register space. In OpenCL, we can declare thread local static arrays to allocate space in the register file. A simple approach is for each thread to load its own data from global memory into its on-chip space directly. While this approach works, memory requests of all threads exhibits strided (hard to coalesce in hardware) access pattern, which is not efficient on modern GPUs.

Another approach to load data into the register space is via local memory. First, data is loaded from global memory in fully coalesced way into the local memory. After this each thread read the row corresponding to itself to its local register space. While this approach fully exploits off chip memory bandwidth, the kernel implementation is relatively complex (Section 3.3.3).

Since the 2D matrix is partitioned in a column major way, each thread needs to access both the register file and the local memory

```
__kernel void StridedStreamScan(__global VEC_TYPE *src, __global volatile int *I, __global VEC_TYPE *dst)
{
    int gid = get_group_id(0), lid = get_local_id(0),c, p=0,src_id;
    __local int C[GROUP_SIZE],  s;
    __local VEC_TYPE lm[GROUP_SIZE][L];
    VEC_TYPE re[R],temp;
    src_id = gid * GROUP_SIZE * (R+L) + lid * (R+L);
    for(i=0;i<L;i++){
        temp = src[src_id+i];
        p = SCAN_ON_VECTOR(temp, p);
        lm[lid][i] = temp;
    }
    for(i=0;i<R;i++){
        re[i] = src[src_id + i + L];
        p = SCAN_ON_VECTOR(re[i],p);
    }
    C[lid] = p;
    barrier(CLK_LOCAL_MEM_FENCE);
    EXCLUSIVE_TREE_BASED_SCAN(C, lid);
    if(lid==0)   c = C[GROUP_SIZE - 1];
    ADJACENT_SYNCHRONIZATION( gid, lid, I, c, &s);
    for(i=0; i<R; i++){ dst[src_id + i + L] = re[i] + s + C[lid]; }
    for(i=0;i<L;i++){
        temp=lm[lid][i];
        dst[src_id + i] = temp + s + C[lid];
    }
}
```

VEC_TYPE: vector type, such as int4, int8.

GROUP_SIZE: The number of threads in a workgroup.

R: The number of the used vector registers per thread.

L: Used local memory size per thread.

SCAN_ON_VECTOR (): Scan on all components of the vector. The second parameter is the propagation value. Return the reduction value.

EXCLUSIVE_TREE_BASED_SCAN(): Perform a exclusive scan on the auxiliary array C using the tree-based scan algorithm.

ADJACENT_SYNCHRONIZATION(): Fetching the accumulation result of previous workgroups from the intermediate array I to s, and write c to I.

Figure 7. Strided StreamScan (Access the global memory in strided way).

to reduce or scan a row of the 2D matrix. As discussed earlier, inter-block synchronization happens after the first reduce phase of each workgroup. One possible optimization here is for each thread to reduce in place while loading data into its local register array.

3.3.3 Memory efficiency VS. Kernel complexity

As discussed in section 3.3.1, in order to access global memory in coalesced manner, regardless of the matrix dimension (2D or 3D), the access order for producing data to and consuming data from the local memory matrix is different. Specifically, all threads write the same row of the matrix at the same time while producing data, and all read the same column at the same time while consuming data. Introducing register space to further expand on-chip

working set size further complicates this issue. In order to access the global memory in coalesced manner, data must be loaded into local memory first and then transferred into the register space. However, since local memory space is much smaller, such data transfers may happen multiple times. Arranging local memory data in 3D manner eliminates local barriers between producing data to and consuming data from local memory buffers. This is more useful if the register space is used, because of the increased memory traffic between global memory and the register file through local memory. However, both optimizations are not free. They increase the OpenCL kernel implementation complexity accordingly.

Note that in the Reduce and Scan computation phases, each thread within a workgroup processes only one row of the matrix. The simple approach is to let each thread read its own portion of data, resulting in strided global memory access patterns. This is much less efficient than coalesced access, but there is no access order discrepancy. Therefore it is not necessary either to arrange data in 3D manner. Also there is no need to transfer data into registers through local memory.

This is the delicate tradeoff between memory efficiency and implementation complexity. Does the improved global memory access speed outweigh the overhead of more complex implementation? We will explore this in more detail in the experimental evaluation section.

3.3.4 Putting it all together

We implemented two families of intra-block scan algorithms, with varying considerations on memory efficiency and kernel complexity. The one with smaller overhead uses strided memory access using register as transmit buffer. The one most sophisticated algorithm takes advantage of the coalesced memory access, but using local memory as transmit buffer. In this paper, we use *Strided* and *Coalesced* to denote the two algorithms outlined above, respectively. Details for *Strided* and *Coalesced* algorithms are shown in Figure 7and Figure 8, respectively.

Another problem is the potential deadlock due to the unguaranteed runtime scheduling order of workgroups. The underlying assumption of this concern is that we distribute the work of input sequence based on the static workgroup ID defined by the static function "get_group_id()". To solve this problem we can allocate all workgroup IDs dynamically. Work distribution among all workgroups is done based on this dynamically allocated runtime workgroup ID. Figure 9 illustrates the idea. Compared to the static allocation mechanism, performance overhead of the dynamic approach is less than 2%.

3.4 Auto-tuning framework

We implemented StreamScan with OpenCL. We designed an auto-tuning framework to search the parameter space automatically. It explores all possible parameter configurations to generate optimal implementations for each algorithm on both NVIDIA and AMD platforms. Table 2 shows all tunable parameters in our auto-tuning framework. These parameters are not independent. For each kernel, the framework explores the entire space for all possible combinations of the rest parameters. A common optimization while loading from global memory is using vectors. This vector length is also tunable. The group size parameter selects optimal dimensions for the 3D matrix, as discussed in section 3.3.1. For each parameter we list the possible values in table 2, but note that the actual value assignment depends on resource constraints of specific GPU platforms. The tuning process starts with initialization, which includes basic configuration of the OpenCL environment and the parameter space. Then for each kernel, it

```
__kernel void CoalescedStreamScan(__global int *src, __global volatile
int *I, __global int *dst)
{
    int gid = get_group_id(0), lid = get_local_id(0);
    __local int lm [BUNCH_NUM][BUNCH_SIZE][LOCAL_SIZE];
    __local int C[GROUP_SIZE], s;
    int re[R*BUNCH_SIZE],i=0, p=0, c;
    for(i=0;i<R; i++){
        LOAD_GLOBAL_ TO_LOCAL(src, lm, gid, lid, i);
        LOAD_LOCAL_TO_REGISTER ( lm, re,  lid ,i); }
    // the reduction value stores in p.
    p = ROW_REDUCE_ON_REG (re,  lid, R, p);
    for(i=0;i<L;i++)
        LOAD_GLOBAL_TO_LOCAL(src,lm,gid, lid,i+R);
    //p: input is the previous reduction value. return reduction value.
    p = ROW_REDUCE_ON_LOCAL (lm,  lid, L, p);
    C[lid] = p;
    barrier(CLK_LOCAL_MEM_FENCE);
    EXCLUSIVE_TREE_BASED_SCAN( C, lid);
    if(lid==0)   c = C[GROUP_SIZE - 1];
    ADJACENT_SYNCHRONIZATION( gid, lid, I, c, &s);
    p=ROW_SCAN_ON_REG (re,  lid, R, c, s, p);
    p=ROW_SCAN_ON_LOCAL (lm,  lid, L, c, s, p);
    for(i=0;i<L;i++)
        PUSH_LOCAL_TO_GLOBAL(lm, dst, gid, lid,i+R);
    for(i=0;i<R;i++){
        PUSH_REGISTER_TO_LOCAL(re,lm,lid,i);
        PUSH_LOCAL_TO_GLOBAL(lm, dst,gid, lid,i);  }
}
```

BUNCH_NUM: The number of thread groups in a workgroup.

BUNCH_SIZE: The number of the threads in a group.

GROUP_SIZE: The number of the threads in a workgroup.

R: The number of register pieces. L: The number of local memory pieces.

LOCAL_SIZE: L * BUNCH_SIZE + 1.

LOAD_GLOBAL_TO_LOCAL(): Load data from global memory to the local memory and organized as a 3D matrix; Each thread group in charge of a flat.

LOAD_LOCAL_TO_REGISTER(): Load data from local memory to the registers.

ROW_REDUCE_ON_LOCAL(): Perform a reduction on every local row.

ROW_REDUCE_ON_REG (): Perform a reduction on every register row.

ROW_SCAN_ON_LOCAL(): Perform a scan on every local row.

ROW_SCAN_ON_REG (): Perform a scan on every register row.

EXCLUSIVE_TREE_BASED_SCAN(): Perform a exclusive scan on the auxiliary array C using the tree-based scan algorithm.

ADJACENT_SYNCHRONIZATION(): Fetching the accumulation result of previous workgroups from the intermediate array I to s, and write c to I.

PUSH_REGISTER_TO_LOCAL(): Push the data in registers to the local memory, each thread in charge of a row.

PUSH_LOCAL_TO_GLOBAL():Push the data in local memory to global memory, each thread in charge of a column.

Figure 8. Coalesced StreamScan (Access the global memory in coalesced way)

```
int lid = get_local_id(0),  gid;
__local int gid_;
if(lid == 0) gid_=atom_add(S,1);  // S is initialized to 0 in global memory
barrier(CLK_LOCAL_MEM_FENCE);
gid=gid_;
```

Figure 9. Dynamical allocation of runtime workgroup ID.

enumerates the parameter space and generates an OpenCL kernel instance. Compilation may fail due to illegal parameter combinations. If compilation succeeds, the tuning process runs the kernel, records the performance statistics, and updates the optimal parameter configuration if necessary. The whole tuning process finishes when the entire parameter space has been explored.

4. Experimental results

In this section, we present experimental results and the analysis of StreamScan. We first discuss our experimental environments, which include one AMD GPU and one Nvidia GPU. Next we report all our performance results and insights.

4.1 Experimental setup

In order to experiment the implementation space on both NVIDIA and AMD GPUs, we implemented StreamScan with OpenCL. For performance comparison purposes, we use two open source scan implementations: CUDPP_Scan [19] and Merrill_scan [13, 20]. CUDPP_Scan is the most popular open source scan implementation. And to the best of our knowledge, Merrill_Scan is the fastest open source scan implementation. We also implemented the 2D MatrixScan [21] within our auto-tuning framework.

We evaluated StreamScan on two platforms, namely NVIDIA Tesla C2050 and AMD Radeon HD5850. We varied the input array size from 1M 4-byte elements to 64M 4-byte elements to experiment a wide range of parameter space. Table 3 lists important system parameters for the two GPU platforms.

Tesla C2050 has hardware ECC support. Unless stated explicitly, all our execution times in this work assume enabled ECC, since this reflects true production running environments. There is no such issue on AMD HD5850 because it does not have ECC support.

4.2 Results

4.2.1 Memory efficiency VS. Kernel complexity

Table 4 shows optimal parameters, and corresponding OpenCL kernel execution time for two algorithms on both hardware platforms. The problem size is 16M(4-byte elements).In our experiments, we varied the problem size from 1M to 64M.The basic trend for optimal parameter configurations across all problem sizes on the same platform is similar.

On Tesla C2050, coalesced kernel performs much better than strided kernel. This means the performance is very sensitive to global memory access patterns, and relatively less sensitive to kernel complexity. Nvidia GPUs have highly optimized mechanisms for memory request coalescing [22]. The sophisticated scalar instruction pipeline is also capable of discovering parallelism for programs with irregular control flow.

On AMD HD5850, strided kernel performs better (7.6%) than coalesced kernel. The system is more sensitive to kernel complexity than global memory access patterns. The main reason is that AMD GPUs employ VLIW design for the instruction pipeline.

Table 2. Tunable parameter space.

Name	Value Domain
Kernel	StridedStreamScan CoalescedStreamScan
Workgroup Size (WS)	64, 128, 256, 512, 1024
Vector Length (VL)	1,2, 4, 8, 16
Group Size (GS)	8, 16, 32, 64
Local Memory Size (LMS)	0K~32K
Register Size (RS)	0K ~ 142K

Table 3. System configurations.

GPUs	Tesla C2050	HD 5850
Operation System	Ubuntu 9.04	Ubuntu 10.10
Runtime	CUDA 4.0	AMD SDK 2.6
Compute Unit (CU)	14	18
Cores	448	288*5
Flops	1.03TFlops	2.09TFlops
Global Memory	3GB GDDR5	1GB GDDR5
Theoretical bandwidth	144GB/s	128GB/s
Local memory banks	32	32
Local memory limits/CU	48KB	32KB
Register file size/CU	128KB	256KB

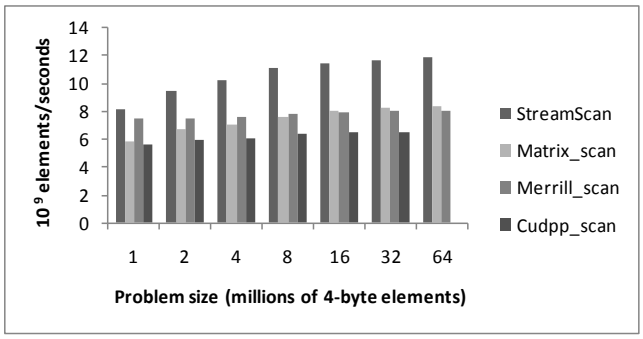
Table 4. Optimal parameters on two platforms.

Parameters		Tesla C2050	HD5850
Coalesced Kernel	WS	128	256
	GS	32	16
	VL	2	1
	LMS	16KB	16KB
	RS	16KB	48KB
	Kernel Time	1.42ms	1.83ms
Strided Kernel	WS	256	128
	VL	4	4
	LMS	12KB	26KB
	RS	4KB	46KB
	Kernel Time	2.43ms	1.71ms

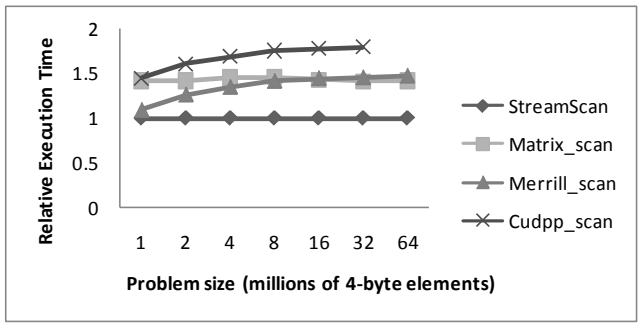
The optimal on chip problem size (RS+LMS) for C2050 is much smaller than HD5850. Smaller size means more workgroups, which in turn implies more adjacent synchronization. It seems sync overhead on C2050 is less prominent than HD5850.

4.2.2 Performance results

Here we compare performance results of StreamScan with prior fast scan implementations on both platforms. Figure 10.a shows the throughputs (measured as 10^9 elements/sec.) of different sequence lengths on NVIDIA Tesla C2050 (CUDPP_Scan can't support 64M elements). As Figure 10.b shows, StreamScan has achieved notable performance speedups. Specifically, when the

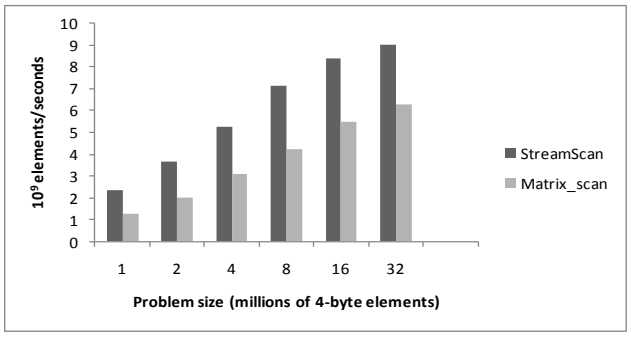


a) Throughput for different problem sizes

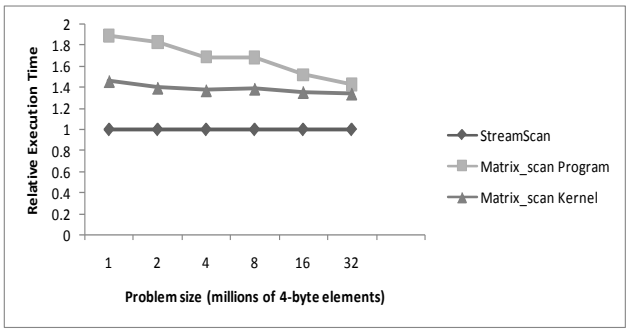


b) Relative execution time for different problem sizes.

Figure 10. Performance on NVIDIA Tesla C2050.



a) Throughput for different problem sizes



b) Relative execution time for different problem sizes.

Figure 11. Performance on AMD Radeon HD5850.

problem size is 16M, speedups of StreamScan, compared to CUDPP_Scan, Merrill_scan and MatrixScan, are 1.8x,1.46x and 1.41x, respectively. There are two reasons for this. First, compared with all other algorithms, StreamScan only needs only ~2N global memory accesses. Second, optimizations like 3D data arrangement and register space exploitation further boost the performance.

Figure 11 shows the test results on AMD Radeon HD 5850 for various problem sizes (Because of the global memory limits, we can't test the 64M 4-byte input sequence on this GPU). Since CUDPP_Scan and Merrill_Scan only have CUDA versions, we implemented MatrixScan in OpenCL for comparison. Figure 11.a shows throughputs (measured as 109 elements/sec.) of different sequence lengths on this platform. Because of high kernel launch overhead, the throughput on AMD Radeon HD 5850 lower much than on NVIDIA Tesla C2050 when the problem size is small. Since AMD GPUs have relatively large kernel launch overheads (~0.3ms), we measure both the kernel execution time and total program execution time in our experiments. As Figure 11.b shows, compared to the program execution time, the speedups at problem sizes of 1M and 32M are 1.9x and 1.4x, respectively. Compared to the kernel execution time, the speedups at 1M and 32M are 1.46x and 1.35x, respectively.

In our experiments, the OpenCL kernel launch overhead on NVIDIA GPUs (~0.02ms) is approximately fifteen times lower than AMD GPUs (~0.3ms). Recall that CUDPP_Scan, Merrill_Scan and MatrixScan all require global barriers. If implemented with kernel invocations, as been studied extensively in previous works [19, 20, 21], the kernel launch overhead may cause notable performance loss on AMD GPUs. Our StreamScan algorithm addresses this issue neatly by eliminating global barriers completely.

4.2.3 Speedup breakdown analysis

As discussed in the previous section, three factors contribute to the performance speedup, namely reduction of off chip I/O traffic from at least 3N to 2N, 3D on chip data arrangement, and register space exploitation. Figure 12 illustrates a further breakdown of relative contributions of these three factors to the overall performance speedup.

On Tesla C2050, among the overall performance speedup, contributions of traffic reduction, 3D arrangement, and the register optimization are 48%, 6%, and 46%, respectively. As can be seen, using register space is a very important optimization. The contribution of thread grouping is relatively small, but still useful. Optimizations on AMD Radeon HD5850 are even more important for performance improvement. As discussed in section 4.2.1, the optimal configuration requires 72KB on chip storage to hold the working set. Shrinking the working set size to smaller than local memory size (32KB) hurts the performance dramatically. Note that for strided kernel, thread grouping is not necessary. In order to illustrate contributions of all three factors, Figure 12 shows relative contributions of three factors for the best coalesced kernel. On Radeon HD5850, among the overall performance speedup, contributions of traffic reduction, 3D arrangement, and the register optimization are 22%, 7%, and 71%, respectively.

4.2.4 Working Set Sensitivity Analysis

For StreamScan, there is another important tradeoff between the working set size (on chip problem size) and the performance. Figure 13 shows the throughput (measured in 10^9 elements/second) variations with different working set size (measured in KB) on Tesla C2050. There are three factors underlying this tradeoff. The smaller the working set size, the faster the intra-block scan. However, as discussed in section 3.1, small working

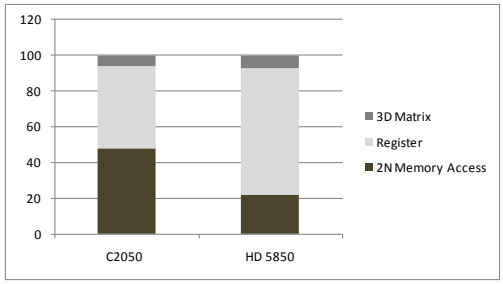


Figure 12. Speedup breakdown.

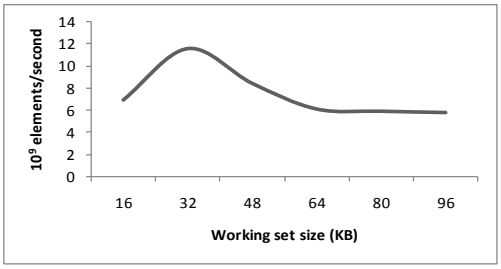


Figure 13. Throughput results of varying working set sizes on NVIDIA Tesla C2050 (Problem Size: 16M).

Figure 14. Throughput results of varying working set sizes on AMD HD 5850 (Problem size: 16M).

set also implies more inter-block synchronization overheads. There is one more resource constraint. Tesla C2050 only has 48KB local memory and 128KB register space per compute unit. The larger the working set, due to resource constraints, the less the exploitable dynamic parallelism between workgroups. The sweet spot is 32KB, with 16KB of local memory and 16KB of register space.

Figure 14 shows the tradeoff between throughput and working set size on AMD HD5850. The profile is very different. Compared to Tesla C2050, each compute unit of AMD HD5850 has much larger register space (256KB), but smaller local memory size (32KB). The implications are twofold. First, the optimal working set size is 72KB, with 26KB of local memory and 46KB of register space (as on-chip cache), due to the large register file. Second, while there are variations, the throughput maintains approximately at the same level when the working set is less than 120KB. There is a balance between the inter-block synchronization overhead and intra-block scan performance when working set size varies. Performance drops sharply when the working set goes beyond 120KB, because register consumption is so high that the system has to spill some registers to global memory.

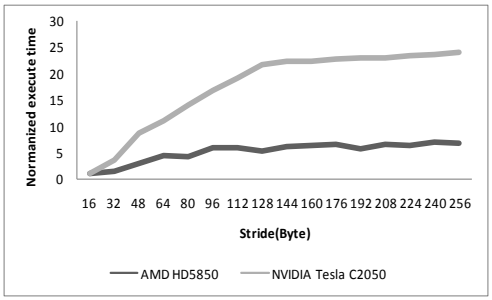


Figure 15. Execution time for different stride sizes (Normalized to coalesced memory access).

4.2.5 Memory system sensitivity analysis

As discussed in previous sections, while Tesla C2050 is more sensitive to memory system performance, AMD HD5850 is more sensitive to kernel complexity. A legitimate question to ask is, for each platform, how much is the performance gap between the coalesced memory access pattern and the strided pattern? Figure 15 shows this result for both GPUs.

For each platform, Figure 15 shows the I/O performance for different stride sizes. The test kernel simply loads 64MB of data from global memory and stores the data back to global memory. For each platform, the execution time is normalized to the performance of the coalesced kernel. As can be seen clearly, the performance gap between these two access patterns is much more prominent on Tesla C2050 than AMD HD5850. This, combined with AMD's VLIW instruction pipeline design, further explains why AMD HD5850 favors the strided kernel.

5. Related works

The scan problem has been studied for decades. Iverson [3] first proposed the problem in APL. Kogge and Stone[4] designed a work inefficient parallel prefix network, which requires exactly $N\log_2 N-(N-1)$ binary operations. Brent and Kung[5] improved this result by devising a tree-based strategy with only $2N - \log_2 N - 2$ binary operations. Horn [27] first implemented the scan algorithm on GPUs. This implementation used the streaming model for the scan operation. Similar to Brent's approach, Sengupta et al [25] first implemented a work-efficient and step-efficient algorithm for scan on GPUs. Harris et al [28] proposed detailed steps on how to implement scan with CUDA. Yuri Dotsenkoet et al [21] first proposed the matrix based scan algorithm on GPUs. The matrix based approach is much more efficient than tree based algorithms. They also proposed the three-phase Reduce-Scan-Scan approach for scan implementation. Duane G. Merrill [13] studied various scan implementation alternatives. In his thesis, he proposed three scan algorithms on GPUs and implemented these three algorithms as an open source library (B40C). In this paper we refer it to Merrill_Scan. Jens Breitbart [24] proposed a scan algorithm on GPUs with a fixed number of threads. Nan Zhang [14] proposed a novel parallel scan for multicore processors. In this work, the input sequence is first divided into several blocks, then one thread is assigned to scan a block. Compared to previous scan algorithms on multicore processors, this work did not require the number of processors to be a power of two. Shucai Xiao et al [26] proposed a fast global barrier implementation on GPUs. Different to them, we proposed the adjacent synchronization to replace global barriers.

6. Conclusion

For all existing GPU scan algorithms, there is a serial dependence between adjacent workgroups. In this work, we've shown that this can be exploited to eliminate global barriers by introducing adjacent synchronization between consecutive thread blocks only. We believe that the idea could be generalized to other algorithms with similar dependence structures.

Based on the adjacent synchronization, we propose StreamScan, a novel scan algorithm on both Nvidia and AMD GPUs. Compared with current state of the art, StreamScan eliminates global barrier synchronization completely. This reduces the global memory I/O traffic from at least 3N to only ~2N. On top of this, we propose thread grouping and register optimization to further boost the performance. Our experimental results show promising speedups on both Tesla C2050 and AMD HD5850.

Besides propose the adjacent synchronization and design a novel scan algorithm on GPUs, our work reveals very different optimization tradeoffs on the two hardware platforms. For Nvidia GPUs, coalesced memory access is critical to performance speedups. Due to the scalar micro-architecture, less compiler and programmer effort is necessary to exploit parallelism. Compared to memory efficiency, optimization efforts on AMD GPUs are more sensitive to kernel complexity. This is largely due to VLIW design of the instruction pipeline. The large register space is very useful for expanding on-chip working set size, which proves to be critical for performance speedups.

Acknowledgments

We would like to thank Yan Li for his collection of materials in the early stage of our work. We thank all the anonymous reviewers for their valuable comments. This paper is supported by National Natural Science Foundation of China (No.61133005, No.61272136, No.61100073), the National High-tech R&D Program of China (No.2012AA010902, No. 2012AA010903). Dr. Guoping Long is supported by National Natural Science Foundation of China (Grant No. 61100072)

References

[1] Blelloch, G. E. Scans as Primitive Parallel Operations. IEEE Trans. Comput, 1989.

[2] Blelloch, G.E. Prefix Sums and Their Applications. Synthesis of Parallel Algorithms, 1990.

[3] Iverson, K. E. A Programming Language. AIEE-IRE '62, 1962.

[4] Kogge, P. M. and Stone, H. S. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. IEEE Trans. Comput, 1973.

[5] Brent, R. P. and Kung, H. T. A Regular Layout for Parallel Adders. IEEE Trans. Comput, 1982.

[6] D. Merrill, and A. Grimshaw Revisiting Sorting for GPGPU Stream Architectures. Tech. Rep. CS2010-03, Department of Computer Science, University of Virginia, 2010.

[7] Cederman, D. and Tsigas, P. On Sorting and Load Balancingon GPUs. SIGARCH Comput. Archit. News, 2008.

[8] D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters, 2011.

[9] Cederman, D., and Tsigas, P. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. Journal of Experimental Algorithmics. 2009.

[10] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place sorting with cuda based on bitonic sort. PPAM 09: Proceedings of the International Conference on Parallel Processing and Applied Mathematics, 2009.

[11] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In Proc. Int'l Symposium on Parallel and Distributed Processing (IPDPS), 2009.

[12] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12), 2012.

[13] D. Merrill. Allocation-oriented Algorithm Design with Application to GPU Computing. PhD thesis, University of Virginia, 2011.

[14] Nan Zhang. A Novel Parallel Scan for Multicore Processors and Its Application in Sparse Matrix-Vector Multiplication. Parallel and Distributed Systems, IEEE Transactions, 2012.

[15] Markus Billeter, Ola Olsson and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures, Proceedings of the Conference on High Performance Graphics, 2009.

[16] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In Proc. of High Performance Graphics, 2009.

[17] Zheng Wei, Joseph JaJa. Optimization of linked list prefix computations on multithreaded GPUs using CUDA. In Parallel & Distributed Processing (IPDPS), 2010.

[18] Mark Harris. State of the Art in GPU Data-Parallel Algorithm Primitives. Tech. Rep. GPU Technology Conference, 2010.

[19] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. http://gpgpu.org/developer/cudpp.

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

[21] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. In ICS'08: Proc. 22nd Annual International Conference on Supercomputing, 2008.

[22] NVIDIA Corporation. Nvidia Cuda C Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[23] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. NVIDIA Tech. Rep, 2008.

[24] Jens Breitbart. Static GPU threads and an Improved Scan Algorithm. In Euro-Par 2010 Work-shop Proceedings, Lecture Notes in Computer Science, 2010.

[25] Sengupta S., Lefohn A. and Owens, J. A work-efficient step-efficient prefix-sum algorithm. Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, 2006.

[26] S. Xiao and W. chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In Parallel & Distributed Processing (IPDPS), 2010.

[27] Horn D. Stream Reduction Operations for GPGPU Applications. In GPU Gems 2, Pharr M., (Ed.).Addison Wesley, ch. 36, pp. 573–589, 2005.

[28] Harris M., Sengupta S. and Owens J. D. Parallel Prefix sum (scan) with CUDA. In GPU Gems 3, Nguyen H., (Ed.).Addison Wesley, ch. 39, 2007.

[29] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, and Shengen Yan. GPURoofline: A Model for Guiding Performance Optimizations on GPUs. Euro-Par 2012 Parallel Processing, 2012.

[30] Khronos OpenCL Working Group, The OpenCL Specification Version: 1.2, 2012.